Artificial Intelligence in Adversarial Games: Papers from the AIIDE Workshop
AAAI Technical Report WS-16-21

Improving Terrain Analysis and Applications to RTS Game AI

Alberto Uriarte and **Santiago Ontañón**
Computer Science Department
Drexel University
{albertouri,santi}@cs.drexel.edu

Abstract

This paper presents a new terrain analysis algorithm for RTS games. The proposed algorithms significantly improves the analysis time of the state of the art via contour tracing, and also offers better chokepoint detection. We demonstrate that our approach (BWTA2) is at least 10 times faster than the commonly used BWTA in a collection of StarCraft maps. Additionally, we show the usefulness of terrain analysis in tasks such as pathfinding and discuss potential applications to strategic decision making tasks.

Introduction

In Real-Time Strategy (RTS) games, the map does not have a fixed configuration like in classic board games such as Chess or Go. Therefore it is a common practice to analyze the map in order to generate a qualitative spatial representation over which to perform reasoning (Forbus, Mahoney, and Dill 2002). This analysis can help us improve tasks such as pathfinding and decision making based on the properties of the terrain. The Brood War Terrain Analysis (BWTA) tool (Perkins 2010) is used by the majority of the participants in the StarCraft AI competitions to analyze the game map. However, this library has very high computational demands, and often fails to detect all relevant areas of a map.

In this paper we present (1) a new algorithm to terrain analysis that, given an input map, generates a set of regions connected by chokepoints (or narrow passages) and (2) an interface for pathfinding queries. Compared to previous related work (Perkins 2010; Halldórsson and Björnsson 2015), our approach is significantly more efficient in terms of execution time and detects chokepoints and corridors more accurately.

Our algorithm is based in finding the medial axis of the map using robust algorithms such as contour tracing and Voronoi diagrams. The proposed algorithm has been packaged into an open-source library[1], which also precomputes common RTS AI tasks like best build locations, closest locations of several points of interest and pathfinding cache.

[1]Source code and binaries can be found at https://bitbucket.org/auriarte/bwta2

The remainder of this paper is organized as follows. First, we provide some background in the context of RTS games and terrain analysis. Then we introduce our algorithm called BWTA2 (Brood War Terrain Analysis 2). Finally, we present our empirical evaluation comparing the map decomposition of our algorithm to previous work and their applications in pathfinding and spatial reasoning.

Background

RTS Games

RTS games are a sub-genre of strategy games where players need to build an economy (gathering resources and building a base) and military power (training units and researching technologies) in order to defeat their opponents (destroying their army and base). One of the particularities of RTS games is that the games are not played always in the same map (or board configuration). That requires an extra step to analyze and understand the properties of each map. While humans learn how to exploit the map in their advantage and recognize potential weak spots, AIs agents need tools to analyze the map in order to perform spatial reasoning. Using bottlenecks as a defending position against larger armies, or detecting higher grounds to reach farther units, are common tactics that humans use all the time. In this paper, we improve the state-of-the-art in map decomposition and spatial tools in order to improve any AI agent. Specifically, we developed a tool to analyze StarCraft maps, a popular RTS game used as a testbed in the AI research community.

Terrain Analysis

Forbus et al. (2002) showed the importance of analyzing game maps to have a qualitative spatial representation over which to perform reasoning. They used a Voronoi diagram to characterize the free space and use it for pathfinding.

Perkins (2010) developed a library called BWTA that using a Voronoi diagram as starting point, decomposes a map into regions and chokepoints. Even if BWTA is one of the most robust libraries, the decomposition process is slow (it takes more than one minute to analyze a map), and often fails to detect some chokepoints.

Higgins (2002) used a different approach than Voronoi diagram and presented the idea of *auras* to expand regions and find chokepoints. Similarly, Halldórsson et al. (Halldórsson

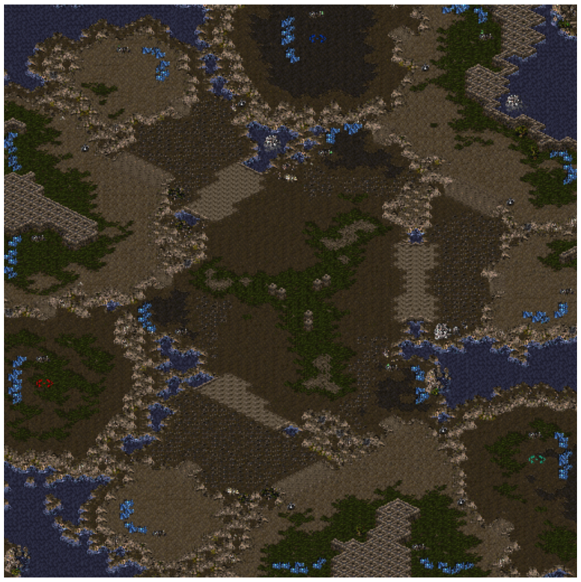
Figure 1: Aztec STARCRAFT map.

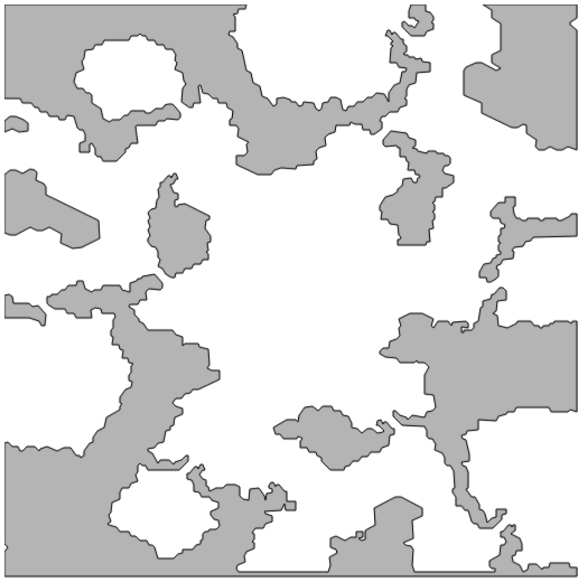
Figure 2: Obstacle polygons detected in gray.

and Björnsson 2015) presented a *water level* approach (which it is discrete) to detect the chokepoints in order to use them as a gateways for the pathfinding heuristic. Although their approach is faster than Perkins, their algorithm tends to generate more false negative and false positive chokepoints.

Bidakaew et al. (2016) presented a map decomposition approach based on finding the medial axis and capable of detecting small corridors (what they call *area chokepoint*), although this approach still detects some false negative/positive chokepoints.

Brood War Terrain Analyzer 2 (BWTA2)

As mentioned, BWTA is used by the majority of STARCRAFT bots. Therefore, we decided to maintain as much as possible the same interface as BWTA to facilitate the migration; and we named our library *Brood War Terrain Analyzer 2* (BWTA2) to reflect the improvement over the previous well known library. Although the algorithm presented in this paper is different from BWTA, the global steps are similar to Perkins' algorithm:

1. Recognize Obstacle Polygons
2. Compute Voronoi Diagram
3. Prune Voronoi Diagram
4. Identify Nodes (Regions and Chokepoints)
5. Extract Region Polygons

For illustrative purposes, we show the result of each step for the STARCRAFT map Aztec, shown in Figure 1.

Recognize Obstacle Polygons

The first goal is: given a raster image made with pixels (a map in our case), extract the polygons of the obstacles. This process called vectorization is a common step in the image processing community. First of all, we need to convert our image into a binary image to be able to detect the relevant shapes we are interested in. To do that, all pixels are marked as *white* if they are walkable by a ground unit, and *black* if they are not walkable. Moreover, there may be undestroyable objects such as neutral buildings that can make a pixel unwalkable. Now, we apply the component-labeling algorithm using contour tracing technique presented by Chang et al. (Chang, Chen, and Lu 2004). This algorithm fulfills two purposes:

1. The contour tracing technique detects the external contour and possible internal contours of each component, i.e., it detects the outer ring and inner rings (or holes) of each obstacle polygon.
2. Each interior pixel of a component is labeled. This will make any query to get what polygon a pixel belongs to a $O(1)$ operation.

Now we have the contour of the obstacles as a sequence of pixels. In order to reduce the number of points that make up the contour we use the Douglas-Peucker simplification (Douglas and Peucker 1973), and after that we iterate over all simplified points and if the distance to the border is less than a threshold (2 pixels in our experiments) we move the point to the border, we call this "anchoring to the border". This last step is to ensure that all the obstacle polygons remain attached to the borders after the simplification. This polygon simplification will help us to reduce the computation complexity of the Voronoi diagram. Figure 2 shows the obstacles detected for the Aztec map.

Compute Voronoi Diagram

In order to improve BWTA, BWTA2 has as secondary goal to reduce the library dependencies, therefore we tried to avoid to use the Computational Geometry Algorithms Library (CGAL) to compute the Voronoi diagram. Instead,

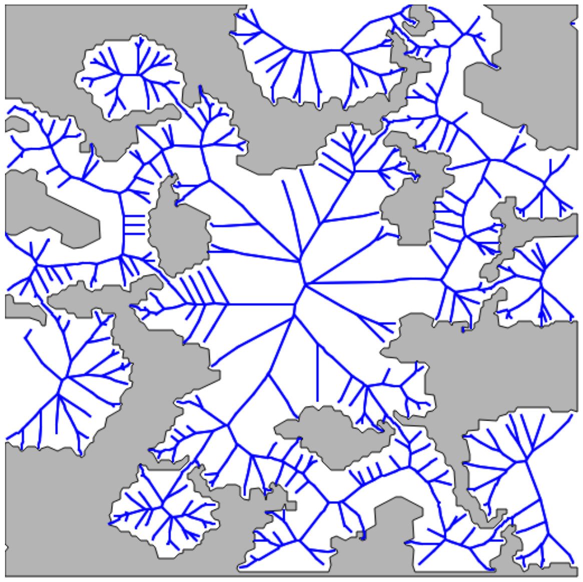
Figure 3: Voronoi diagram of walkable areas.

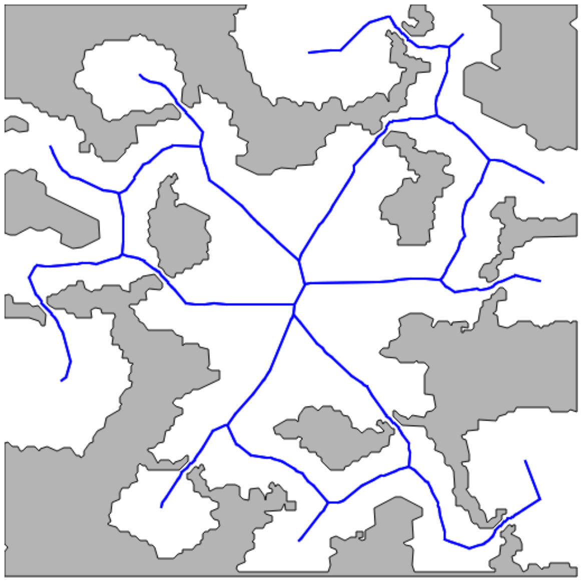
Figure 4: Medial axis from pruning the Voronoi diagram.

we use the implementation in the Boost library (CGAL depends on Boost and other libraries to work). Boost uses Fortune's sweep-line algorithm (Fortune 1986) to compute the Voronoi diagram, while CGAL uses an incremental algorithm (Karavelas 2004) that allows to add segments that may intersect at their interior (with a performance penalty). Since we know all the segments beforehand and all the polygons are ensured to be simple thanks to the polygon simplification step, we can use the sweep-line algorithm that significantly outperforms the incremental one used by BWTA. In order to avoid infinite Voronoi segments, we also add the necessary segments to close the border of the map (not drawn in our Figures). Boost returns the Voronoi diagram as a half-edge data structure. For our purposes we simplify this structure to a regular graph represented by an adjacency list. We only add to the graph the Voronoi points that do not lie inside of an obstacle polygon. Figure 3 shows our computed Voronoi diagram.

Prune Voronoi Diagram

In this step we want to compute the *medial axis* given the Voronoi diagram by removing the "hairs" of the Voronoi diagram. To do this, we compute the distance to the closest obstacle for each vertex in our graph. These queries can be optimized using a R-tree structure (Guttman 1984) with a packing algorithm (Leutenegger, Lopez, and Edgington 1997). Adding all the segments to the R-tree will let us perform a query to know the closest obstacle of a point, let us call this the *radius* of the vertex. Now, we use all the leaf vertices in the graph to initialize the list of candidate vertices to be pruned. Each vertex in the list is evaluated once, if the vertex is too close to an obstacle (has a radius less than 5 pixels) or if the parent is farther to an obstacle, we remove the vertex and the edge. After removing the edge, if the parent vertex becomes a leaf, we add it to the list of candidate vertices to be pruned. Figure 4 shows how the resulting medial axis still captures the structure of the map but with less complexity.

Identify Nodes (Regions and Chokepoints)

Now we identify all local maxima, i.e., vertices whose radius is bigger than any of their neighbors. Notice that this corresponds to the biggest inscribed circle of a region; and all local minima, i.e., vertices whose radius is smaller than its neighbors, are chokepoints of the map or narrow passages. Our proposed algorithm only iterates once over each vertex, and therefore it has a $O(|V|)$ time complexity. Starting from a leaf vertex, we mark that vertex as a region node (after the pruning, all remaining leaf vertices are local maxima) and we add its children to the list of vertices to explore. Then for each vertex in the list we proceed as follows:

1. We add all unvisited children to the list of vertices to explore.
2. If the vertex has a degree other than two, we mark it as a *region node* since it is a leaf or an intersection point.
3. When a vertex has two children and it is a local minima, if the parent is also a local minima we mark as a *chokepoint node* only the vertex with the smallest radius, otherwise we mark the current vertex as a *chokepoint node*.
4. In the other hand, when a vertex with two children is a local maxima, if the parent is a local maxima we mark as a *region node* the vertex with the biggest radius, otherwise the current vertex is marked as a *region node*.

The next step is to simplify our graph, first we create a new graph with only the marked nodes (region or chokepoint) and their connections; and then since all paths must alternate between a chokepoint and a region node, all connected regions (those that are intersections with another region) are merged, keeping only the region with the biggest radius. Our presented algorithm is more efficient than the one presented

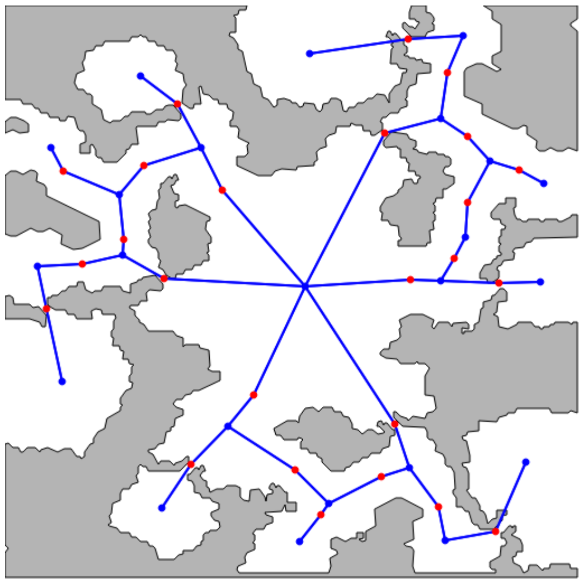

Figure 5: Final nodes detected (chokepoints in red and center of regions in blue).

by Perkins since we only visit the vertices $|V| + |V| + |N|$ (the first time to mark the nodes, the second time to simplify the graph to N nodes, and the third time to merge consecutive regions). Figure 5 shows the region nodes (blue dots) and chokepoints nodes (red dots) identified by our algorithm.

Extract Region Polygons

Once the chokepoints nodes have been identified, we use the previous R-tree structure to do an iterative query to find the two obstacle points that crosse each chokepoint node. Now, to generate the region polygons we start with a square covering the whole map and we compute the difference with the obstacle polygons to generate the walkable polygons. And finally we compute the difference of walkable polygons and the chokepoints segments to split them into the region polygons. Figure 6 shows the final region division. As a final step, we generate our internal data structure where each *Region* has its own polygon, the center point from the region node (notice that this position is ensured to fall inside the polygon in opposition to a centroid of a polygon), and a set of adjacent *Chokepoints*. And each *Chokepoint* has a segment, a middle point and two connected *Regions*.

Extra cache operations for AI queries

BWTA2 performs several extra computations to speed up common operations that most AI agents use.

- A component-labeling algorithm is performed to the walkable pixels in order to know what regions are connected and which ones are islands.
- We identify all potential locations to build a base. To optimize resource gathering, bases are located equidistant to a cluster of resources. We identify cluster of resources using a DBSCAN algorithm (Ester et al. 1996) with the following constraints: the minimum distance to a group resources is 6 build-tiles, the minimum resource cluster size to be considered is 3. After that, for each cluster we find the build location that it is equidistant to each resource in the cluster.

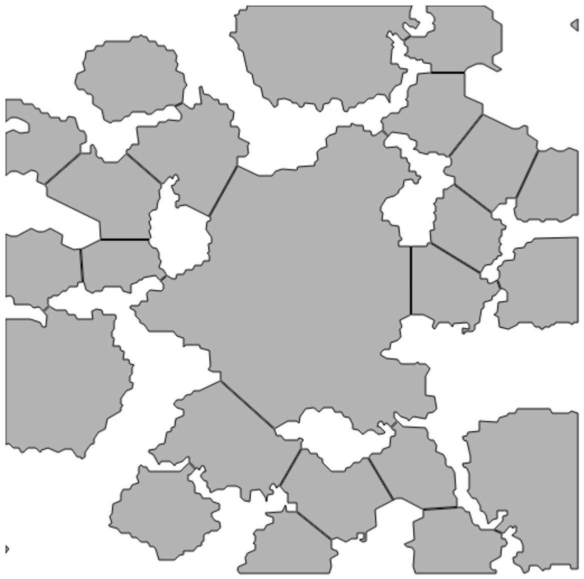

Figure 6: Map decomposed in regions.

- Despide decades of research, pathfinding is still an expensive computation and an open research area. BWTA2 implements HPA* (Botea, Mueller, and Schaeffer 2004) which uses the concept of *rooms* and *gates* as a higher-level structure to perform the search. Here, we use our division of *regions* and *chokepoints* as an abstraction for our search. The idea of using smart terrain decomposition for pathfinding was also explored by Halldórsson et. al. (2015). One of the major speedups of the algorithm is achieved by caching the distance between all *gates* (or *chokepoints* in our case).
- Closest point of interest. A common query is the distance to a closest object (base location, chokepoint, ...). A *multi-seed flood-fill* algorithm is performed for each relevant object. The idea of a *multi-seed flood-fill* algorithm is to initialize a FIFO list with the different seed positions and use this list to pop the next position to check and push the following positions to explore; we also customized it to increase the cost once a position switches from walkable to unwalkable for the first time. Hence, the result shows the closest object for a ground unit. Figure 7 shows the closest distance to a base location.

Experimental Evaluation

We compare the algorithm presented in this paper (BWTA2) against BWTA, since this is the only open source state-of-the-art algorithm. We analyzed 3 different popular maps used in the STARCRAFT AIIDE competition that cover different map size ranges: Heartbreak Ridge, Benzene and

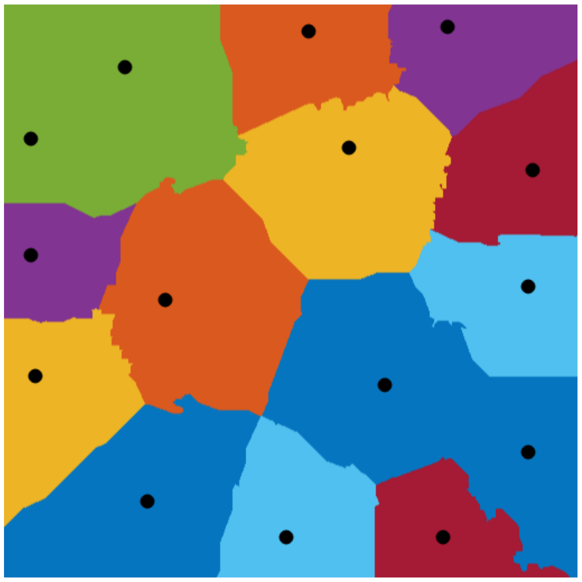

Figure 7: Multi-seed flood-fill distance to closest base location.

Table 1: Time in seconds to analyze each map by BWTA and BWTA2.

Map Name	Map Size	BWTA	BWTA2
Destination	128x96	54.15	3.53
Heartbreak Ridge	128x96	53.55	4.32
Benzene	128x112	55.82	5.87
Aztec	128x128	66.83	5.39

Aztec. We compare the time to analyze the map by each algorithm. Table 1 shows that our proposed algorithm is more than 10 times faster than BWTA, and it only takes less than 6 seconds to analyze complex maps.

In the other hand our approach does a better job finding chokepoints, specially long corridors or small symmetric regions. Figure 8 shows the chokepoints detected by BWTA2 in red (regions in blue are not merged); Figure 9 shows the chokepoints detected by BWTA in red and the missing chokepoints that BWTA2 was able to find in green. As mentioned, the missing points are long thin corridors or small regions with two chokepoints. Since regions are symmetrical, we can observe how sometimes BWTA detects the same chokepoints in the symmetric region and other times it only detects one of the chokepoints.

Applications to Pathfinding

The most common use of terrain analysis is for pathfinding or navigation. As we mentioned there are several algorithms capable of using the information of terrain analysis to guide the pathfinding search. BWTA2 implements the idea of HPA* (Botea, Mueller, and Schaeffer 2004) to use the *chokepoints* as *gates*. Besides the algorithm being near-optimal, in RTS games we can have hundred of units

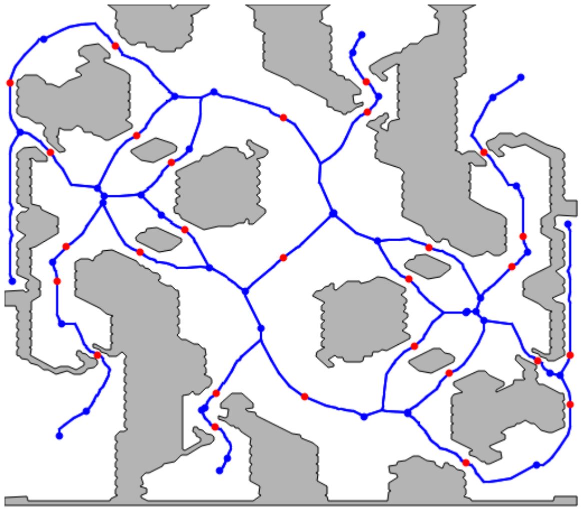

Figure 8: Chokepoints in red detected by BWTA2.

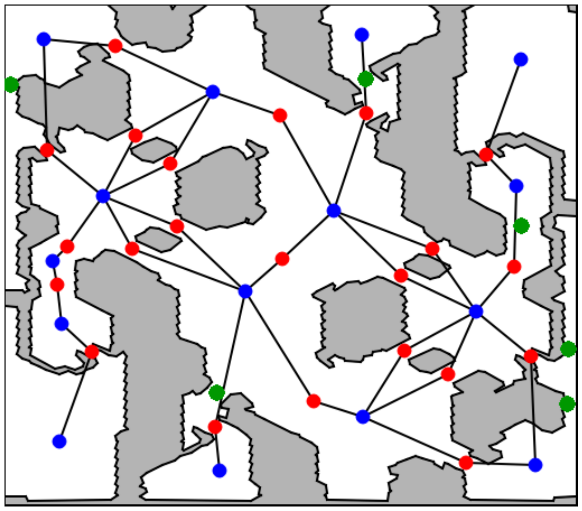

Figure 9: Chokepoints (detected in red, missing in green) by BWTA.

computing distance checks, usually for planning a long-term plan, therefore a fast approximation like HPA* is perfect for our case. Keep in mind that the environment in RTS games is dynamic (i.e., changes can occur while units are moving), the navigation system should be combined with a reactive one capable of handling the frame-by-frame unit control. Here we are only considering path planning. Table 2 shows a time comparison in ms between A* and HPA* implemented in BWTA2, showing that HPA* can be 64 times faster than A*; a significant difference, specially important in games with a lot of units moving in real-time.

Applications to Strategic Decision Making

Lidén (2002) showed how terrain analysis can help other tasks beyond pathfinding. Chokepoints or waypoints can be used as key points to calculate visibility to perform an intelligent attack positioning such as flanking or squad coordina-

Table 2: Time in milliseconds to compute the distance between bases in Aztec map.

Tile Positions	A*	HPA*
(9,84) and (69,7)	41.13	0.95
(9,84) and (118,101)	46.72	0.73
(69,7) and (118,101)	41.38	1.09

tion.

Additionally, we presented how to detect potential base locations to decide the next base expansion. Quantifying the tactical importance of each region can help establish the most profitable location from the point of view of map control, or which the best enemy base to attack. Finding the closest relevant object is an expensive query executed several times during a game. We cached these queries to improve the overall AI agents performance.

Finally, another application of terrain analysis is game tree search. For example, Uriarte and Ontañón (2014) used a map decomposition as a way to define an abstraction over which to employ Monte Carlo Tree Search.

Conclusions

This paper introduced BWTA2, a new algorithm that improves the state-of-the-art of terrain analysis reducing considerably the computational time of map analysis. Additionally, it produces a better map decomposition detecting more meaningful chokepoints and offers better tools for AI agents. The algorithm has been implemented into an open-source library, available to the research community.

As a part of future work we want to consider unwalkable areas during the analysis, in order to produce safe paths for air units, or help transports make better decisions (a common example is to use an air transport to help ground units to cross a cliff and avoid a strong defended chokepoint).

References

Bidakaew, W.; Sompagdee, P.; Ratanotayanon, S.; and Vichitvejpaisal, P. 2016. Rts terrain analysis: An axial-based approach for improving chokepoint detection method. In *Knowledge and Smart Technology*, 228–233.

Botea, A.; Mueller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1(1):1–22.

Chang, F.; Chen, C.-J.; and Lu, C.-J. 2004. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding* 93(2):206–220.

Douglas, D. H., and Peucker, T. K. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization* 10(2):112–122.

Ester, M.; Kriegel, H.; Sander, J.; and Xu, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Knowledge Discovery and Data Mining*, 226–231.

Forbus, K. D.; Mahoney, J. V.; and Dill, K. 2002. How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems* 17(4):25–30.

Fortune, S. 1986. A sweepline algorithm for Voronoi diagrams. In *Symposium on Computational Geometry*, 313–322. ACM.

Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. In *International Conference on Management of Data*, 47–57. ACM.

Halldórsson, K., and Björnsson, Y. 2015. Automated decomposition of game maps. In *AIIDE*, volume 15, 122–127.

Higgins, D. 2002. Terrain analysis in an rts - the hidden giant. *Game Programming Gems 3* 268–284.

Karavelas, M. I. 2004. A robust and efficient implementation for the segment Voronoi diagram. In *Int. Symp. on Voronoi Diagrams in Science and Engineering*, 51–62.

Leutenegger, S.; Lopez, M. A.; and Edgington, J. 1997. STR: a simple and efficient algorithm for R-tree packing. In *International Conference on Data Engineering*, 497–506.

Lidén, L. 2002. Strategic and tactical reasoning with waypoints. *AI Game Programming Wisdom* 211–220.

Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *AIIDE*. AAAI Press.

Uriarte, A., and Ontañón, S. 2014. High-level representations for game-tree search in RTS games. In *AIIDE*.