Wall Building in the Game of StarCraft with Terrain Considerations

Martin L. M. Rooijackers and Mark H. M. Winands
Games & AI Group, Department of Data Science and Knowledge Engineering, Maastricht University
Maastricht, The Netherlands
mlm.rooijackers@student.maastrichtuniversity.nl, m.winands@maastrichtuniversity.nl

***Abstract*—StarCraft is a Real-Time Strategy game, which has a large state-space, is played in real-time, and commonly features two opposing players, capable of acting simultaneously. One of the aspects of the game is building walls. In this paper, we present an algorithm that can be used for wall building for an agent playing the game of StarCraft: Brood War.**

***Index Terms*—StarCraft, Wall building, Real-Time Strategy game**

I. Introduction

One of the aspects of Real-Time Strategy (RTS) games is wall building. The general strategy of wall building is placing down structures such that you are safe from an attack at a given position. Usually this is done to protect your base and production facilities without having to rely solely on combat units.

This topic is not unique to StarCraft. Previous work has been done in the game Empire Earth [1], where the developers used Graham Scan to decide where to place a wall. A generic wall-building algorithm was also presented in [2].

In StarCraft: Brood War, wall building requires an extra property, which these algorithms do not take into account. Namely, buildings have gaps between them (see Figure 2). Depending on the size of the unit, walls might not be closed off for all units. This sometimes is desirable as well, in case you want your own (small) units to pass through, but do not want the enemy units to pass your wall.

For wall building specifically for StarCraft, there has been an approach that uses answer set programming to add this extra constraint [3]. By specifying the gaps and other constraints through a declarative language, Certicky was able to use the ASP solver clingo to create wall while considering building gaps. This approach has been improved by Richoux *et al.* [4].

The problem with these approaches is that none of them address the gap between buildings and the terrain (mainly cliffs and other natural obstacles). Because of this, some of the walls created with these methods will still have gaps in them that let enemy units through. We propose an algorithm based on the pathfinding algorithm A^* to ensure that the wall from our algorithm will stop whichever enemy unit it is supposed to stop. The advantage of this method is that the constraints are checked by checking if a path exists in the game itself, thus ensuring that a wall found by our algorithm is tight. With an extra calculation step, our algorithm can also create walls where smaller units can pass through, but the larger units of the opponent cannot.

This paper is structured as follows. First, Section II gives the problem definition of wall building. Next, Sections III and IV discuss the wall-building algorithm. Section V describes the pseudocode implementation of the wall-building algorithm. Section VI describes the experimental setup and the results of the experiments. Finally, Section VII draws conclusions from the results and presents future work.

II. Problem Definition

In the RTS game of StarCraft: Brood War, a wall is a set of structures and units placed in such a way that no enemy unit can pass from one side of the wall to the other. In this section we describe the problem that our algorithm tries to solve.

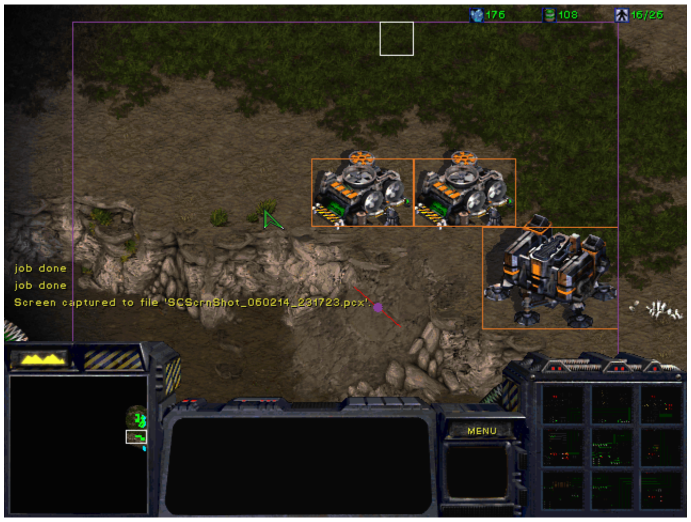
The StarCraft map consists of two grid types: the walk grid, where each cell is an 8×8 pixels square, and the build grid, where each cell is a 4×4 walk tile square (hence 32×32 square of pixels). Some of these build tiles are buildable, and others are not due to natural obstacles (e.g., cliffs) or due to the game rules (buildings cannot overlap).

Each building has a build size and a real size. The build size indicates the height and width of the building in terms of building tiles. For example, a Terran supply depot has a build width of 3 and a build height of 2. The real size indicates how much walkable space the building takes up. In the case of StarCraft: Brood War, the actual space taken up by some of the buildings is less than its build size $\times$ 32 pixels, leaving some additional space for passing it. This causes gaps to be created between buildings placed next to each other and to natural obstacles (cliffs have these gaps as well).

Thus the wall-building problem is about finding locations for buildings such that a given enemy unit (given in pixel width and height) cannot pass through the wall. An extra constraint can be that another smaller unit has to be able to pass through while the given enemy unit still cannot pass through (this allow for hit & run tactics where ranged units can retreat behind a wall after shooting).

The algorithm presented in this paper is specifically for checking if a given set of buildings can satisfy these constraints. The problem of determining the minimum number of buildings required for building a wall is beyond the scope of this paper. This however can be easily added by having an algorithm generating a list of buildings and using the algorithm presented in this paper to check if those buildings can form a wall, until a suitable combination of buildings is found.

III. Approach



job done
job done
Screen captured to file 'SCScrnShot_060214_231723.pcx'.

Fig. 1. Example of a wall found using the wall-building algorithm. Choke point is indicated with a red line and a purple circle.

In our approach we use a wall to seal off a choke point. A *choke point* is a location on the map (terrain) that connects two regions [5] (see also Figure 1). If a wall is built near this choke point, then the two regions of the choke point are separated (no longer reachable by ground units).

Since both buildings and units can be used in a wall, we use the term structure to indicate a part of a wall (either a building or a unit). The first part of the wall-building algorithm requires a way to generate a set of structure locations, which can potentially form a wall. In the game of StarCraft, units and buildings cannot overlap, buildings cannot overlap with unsuitable ground and units cannot overlap with unwalkable ground. We use a depth-first search algorithm to determine possible structure locations. At the initial depth, the algorithm places a structure close to the choke point. After selecting a location for the first structure, the next depth includes a new structure that is placed adjacent to any of the structures already placed. This is done because a wall should not contain a gap. Therefore each building and unit is adjacent to at least one other building or unit. This process continues until all structures have been placed.

Each time that the placement algorithm placed all structures (reached maximum depth), the validation algorithm checks if the placement of the structures forms a wall. Determining if a set of structures forms a wall requires checking if there is a path that goes from one side of the choke point to the other side that passes through the choke point that is supposed to be walled of. For this we compare two different methods. In both methods we limit the search to a 16×16 build grid around the choke point to ensure that all possible paths have to go through the choke point that is supposed to be walled of. The first method that we use is a flood fill starting from one side of the choke point. If the flood fill reaches the other side of the choke point then the algorithm will generate the next set of possible structure locations which can potentially form a wall. The second method is the A^* algorithm. The A^* algorithm tries to reach the other side of the wall based on heuristic search instead of a brute force approach like flood fill.

IV. Wall-Building Algorithm

The wall-building algorithm used to calculate a wall is based on a depth-first search approach to find possible structure placements combined with flood fill or A^* to check if a structure placement is a wall. Just like the declarative programming approach from Certicky [3], we construct a 16×16 grid around a choke point (see Section III for the definition of a choke point). Besides the choke point location, the algorithm also requires the list of buildings and units available to create the wall.

The wall placement algorithm starts off with no structures placed (depth-first search at a depth of 0). At this point (depth) the algorithm will pick one of the possible structures and place it down at a location close to the choke point. Determining a possible location is performed by checking if every tile occupied is buildable (if it is a building) or walkable (if it is a unit). After the initial placement, the wall placement algorithm places the next structure adjacent to the initial structure. This adjacency is 8-ways (horizontal, vertical, and diagonal). Every next structure from this point is then placed adjacent to at least one of the structures already placed. This process continues until all available structures have been placed.

Once all structures are placed (at the leaf node), the algorithm starts a flood fill or A^* from a tile on one side of the wall. Both the flood fill and A^* algorithm try to reach a tile on the other side of the choke point by passing through the choke point. If successful, the current placement of structures is not a wall. In this case the depth-first search backtracks and tries a different placement of structures. Besides walkable gaps on the tiles not covered by the structure, the flood fill and A^* can also pass between gaps formed by buildings (see also Figure 2). Each building has a certain number of pixels as a gap on each side (top, bottom, left, and right). When a side of a building is adjacent to the side of another building, the corresponding sides combine the pixel values.

The algorithm used by Certicky *et al.* [3] did not take into account that two buildings placed diagonally also create a gap. Thus this algorithm sometimes misses a possible wall, because it does not take the gap into account (see also http://wiki.teamliquid.net/starcraft/Walling).

The algorithm from Richoux *et al.* [4] does not take the terrain (cliffs and other natural obstacles) into account. These terrain features have gap values as well. Although the gap values of buildings are known, it is not yet known what the exact gap values of all terrain features are. Therefore, our algorithm uses the walk grid to calculate if a unit can fit through the gap of a building and a cliff or other terrain feature. Even though the gap size is not known, the Brood War API

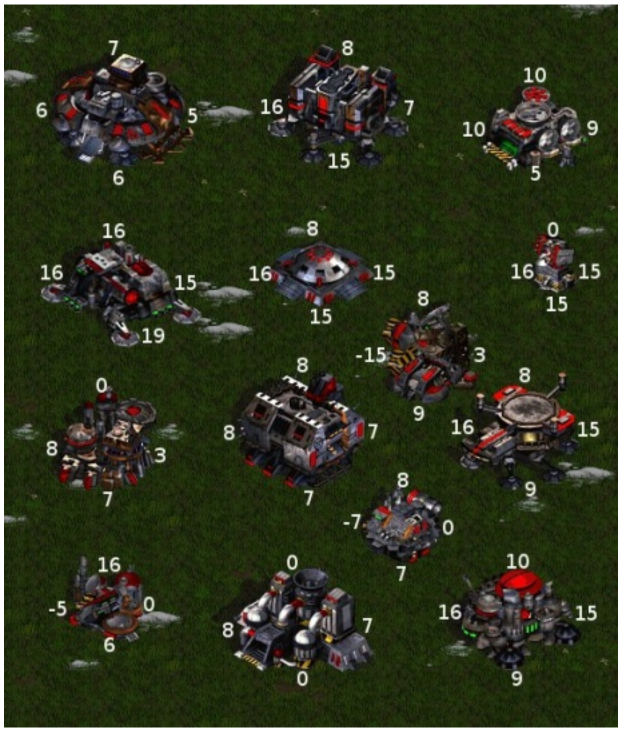


Fig. 2. Gaps between Terran buildings in pixels. The numbers indicate the size of the gaps in pixels. Picture taken from teamliquid wiki.

does give information about which tile can be walked on. Thus instead of calculating the gap between a building and a cliff, we instead take the total number of walkable tiles between a building and a cliff. This is then multiplied by the pixel size of a walkable tile (8×8, see Section II). This estimation is an upper bound to the gap size, so the walls created with this estimation will stop the unit it is supposed to stop. However, if the actual gap is smaller than the estimation, our algorithm might miss a possible wall. Determining the exact gap size when terrain is involved is left for future work.

The exact details of this process is an implementation detail. The code for the wall building can be found at https://github.com/MartinRooijackers/LetaBot.

V. Implementation

The wall-building algorithm requires a choke point, a tile on one side of the choke, a tile at the other side of the choke and a list of structures. The other information about buildable and walkable locations can be derived from the BWAPI. This pseudocode gives a high level overview of the implementation for a StarCraft agent.

The wall-building algorithm requires the following input:

- *S*: a list of structures, each containing its width and height
- *B*: a list of structures already placed, each containing its width and height and (x, y) position
- *choke*: tile location of a choke point
- *sTile*: start location of the flood fill
- *eTile*: end location of the flood fill
- *Enemy*: (width, height) tuple indicating the size of the unit that should not be able to pass through the wall

The algorithm is split into two components: the wall placement algorithm that determines valid structure locations and a wall validation algorithm that checks if these locations form a wall. The second step can be performed by either A^* or flood fill. The wall placement algorithm is given in the pseudocode below (see Algorithm 1). The algorithm starts with calling the WallIn function with the parameters mentioned above. This function calls StructurePlacement, which uses depth-first search to place the structures. As long as a structure needs to be placed, the algorithm uses the ValidLoc function to determine a place to put the building or the unit. This process is repeated until all structures are placed. Once all structures are placed, the algorithm checks whether the structure placement is a wall with the CheckWall function. Two variants of this function have been implemented by either using a flood fill or A^*.

Algorithm 1 WallPlacement
```
procedure WallIn($S, choke, sTile, eTile$)
    StructurePlacement($0, S, \emptyset, choke, sTile, eTile, Enemy$)
end procedure

procedure StructurePlacement($depth, S, B, choke, sTile, eTile, Enemy$)
    if size($S$) $= 0$ then                          ▷ all structures placed
        for all $x \in \{choke_x - 8, \ldots, choke_x + 7\}$ do
            for all $y \in \{choke_y - 8, \ldots, choke_y + 7\}$ do
                Visited($x, y$) $\leftarrow$ False ▷ clear last flood fill
            end for
        end for
        isWall $\leftarrow$ CheckWall($sTile, eTile, Visited, Enemy$)
        if $isWall = false$ then
            return ▷ This is not a wall in, generate a new structure location
        end if
        if $isWall = true$ then return
            Output/Store current structure locations and end algorithm
        end if
    end if
    for all $x \in \{choke_x - 8, \ldots, choke_x + 7\}$ do
        for all $y \in \{choke_y - 8, \ldots, choke_y + 7\}$ do
            if ValidLoc($x, y, S_0, depth$) then
                StructurePlacement($depth + 1, S \setminus S_0, B \cup \{(S_0, x, y)\}, choke, sTile, eTile, Enemy$)
            end if
        end for
    end for
end procedure
```

The ValidLoc function (see Algorithm 2) checks whether a building can be placed at a certain location. Since a wall requires all buildings to be adjacent, this function also checks whether the build location is adjacent to another location already occupied. The only exception is the first building, since it cannot be adjacent to anything yet.

Algorithm 2 ValidLoc

```
procedure ValidLoc($x, y, struct, depth$)
    $Adjacent \leftarrow False$
    for all $xTile \in \{x, \ldots, x + (struct_w - 1)\}$ do
        for all $yTile \in \{y, \ldots, y + (struct_h - 1)\}$ do
            if Occupied($x, y$) $= True$ then          ▷ BWAPI function
                return $False$
            end if
            if Tile adjacent to other structure then          ▷ 8-way
                $Adjacent \leftarrow True$
            end if
        end for
    end for
    if $Adjacent = False \land depth \neq 0$ then ▷ adjacency check
        return $False$
    end if
    return $True$
end procedure
```

There are two ways to implement the CHECKWALL function used in the wall placement algorithm. The first option is to use the flood-fill algorithm. The second option is to use the A^* algorithm. Both algorithms use information from the BWAPI to determine if a position is invalid. If a position is invalid, it cannot be traversed. An invalid position is a position where:

- The *x* or *y* position is outside of the map.
- The *x* or *y* position is outside of the 16×16 grid.
- The gap between buildings is not large enough (see Figure 2).

Hence, a valid position is a position on the map that does not have these characteristics. The flood-fill algorithm uses 8 directional movements. For the implementation of A^*, we use the Manhattan distance heuristic. Since the standard variant of the algorithms are used, the pseudocode is not reproduced here. The implementation details can be found in the source code.

VI. Experiments

A. Setup

In the first experiment of this paper, we investigated the computing time of the wall-building algorithm. For this, we have used the standard "1 barracks + 2 supply depots" to wall off the starting location. This configuration is the standard build order that is used in professional games where the Terran player wants to protect the starting location from rush strategies. We picked four CIG maps from the 2017 tournament and one from the general CIG map pool. The maps we selected are:

- Hitchhiker 1.1
- Tau Cross 1.1
- Neo Aztec 2.1
- Andromeda 1.0
- Python 1.3

We have run the test on each map 20 times. The mean running time of the flood fill and A^* variant of the algorithm can be found in Table I. The table reveals that A^* decreases the computation time considerably. The standard deviation can be found in Table II. It shows that if our algorithm finds a wall, it will do so quickly. However, if a wall does not exist, our algorithm will try all possibilities, which causes the high deviation.

Map/Algorithm	Flood Fill	A^*
Hitchhiker 1.1	11.6s	6.5s
Tau Cross 1.1	2.6s	1.7s
Neo Aztec 2.1	0.6s	0.7s
Andromeda 1.0	4.3s	3.3s
Python 1.3	15.3s	11.4s

TABLE I
RUNNING TIME OF EACH ALGORITHM VARIANT IN SECONDS. AVERAGE OF 20 EXPERIMENT RUNS.

Map/Algorithm	Flood Fill	A^*
Hitchhiker 1.1	11.0s	5.8s
Tau Cross 1.1	1.3s	0.6s
Neo Aztec 2.1	0.07s	0.7s
Andromeda 1.0	2.6s	0.2s
Python 1.3	14.4s	9.5s

TABLE II
STANDARD DEVIATION FROM THE RUNNING TIME OF EACH ALGORITHM VARIANT IN SECONDS.

We have used this algorithm to give our StarCraft agent LETABOT the capability to build a wall in order to stop a rush build. Such rush builds are used by professional StarCraft player and bots. Our wall-building algorithm was first used in the CIG 2014 tournament. It has been used in every major StarCraft AI tournament ever since. The effect of the wall placement is especially notable when our agent plays against a rush build, which is a popular strategy in the StarCraft AI tournament. We got the following notable tournament results with the help from this wall-building algorithm:

- CIG 2014: 3rd place
- CIG 2016: 3rd place
- CIG 2017: 4th place
- AIIDE 2014: 3rd place
- AIIDE 2016: 4th place
- SSCAI 2014: 1st place mixed+student
- SSCAI 2015: 1st place student division
- SSCAI 2016: 1st place mixed+student
- SSCAI 2017: 2nd place student division

In the last series of experiments we also tested what would happen if LETABOT did not use the wall-building algorithm. For this we disabled the wall-building algorithm in LETABOT and put it up against two rush bots (CARSTEN NIELSEN and

WULIBOT) on the 5 maps from the first experiment. The results of that can be seen in Table III, winning only 51% ($\pm 9.8\%$) of the games. If the wall-building algorithm is turned on, our bot scores a 100% win rate against these bots.

What is noticeable, is that the wall-building algorithm does not add much when playing on a large map like "Andromeda", where the size alone makes rush strategies less effective. Maps where you start on the high ground and have a ramp that can be used as a choke point, help in the defensive without walls as well ("Hitchhiker" and "Python"). But with the exception of large maps like "Andromeda", LETABOT benefits from using a wall-building algorithm to deter rush strategies.

Map/Bot	CARSTEN NIELSEN	WULIBOT
Hitchhiker 1.1	4-6	3-7
Tau Cross 1.1	5-5	0-10
Neo Aztec 2.1	10-0	0-10
Andromeda 1.0	10-0	10-0
Python 1.3	7-3	2-8
Total	36-14	15-35

TABLE III
WIN RATE OF LETABOT WITHOUT USING A WALL-BUILDING ALGORITHM (FORMAT: WINS-LOSES).

VII. CONCLUSIONS & FUTURE RESEARCH

In this paper, we have shown two variants of an algorithm that can be used for building walls in StarCraft. Unlike other methods, this algorithm guarantees that a wall can be used to ensure that a given unit cannot pass through it. The downside compared to other methods is that iterating through the possibilities to ensure that the wall is tight, is a costly calculation. The A^* heuristic search improves this, but this algorithm is still mainly recommended to be used for pre-calculating building positions to ensure a tight wall. Because the map of StarCraft is static, this information can be calculated and stored, such that it can be retrieved next game and be used immediately. Thus this algorithm becomes a tool, like the terrain analysis tool BWTA, which is used by our StarCraft agent for choke point analysis and splitting the map in regions.

One of the things still missing from the pathfinding is the exact data on gaps created by the terrain. For now our algorithm used the walkable data given by the BWTA. The walls created by this are tight, but the criteria are stricter than they have to be. Thus our algorithm sometimes report that there is no wall possible, even though one exists. This explain the large variance of running time between maps, since our algorithm takes less time if it finds a wall (because then it can terminate the search). Most of the time, an alternative wall (further away from the starting position) will be found at the cost of extra running time. This is especially the case on maps like "Python".

A way to improve the running time is to have some extra checks in place for the structure placement to reduce the number of placement choices that can be trivially calculated not to be walls.

REFERENCES

[1] T. Teich and I. Davis, "AI Wall Building in Empire Earth II," in *AIIDE*, 2006, pp. 133–135.

[2] M. Grimani, "Wall building for RTS games," *AI Game Programming Wisdom*, vol. 2, pp. 425–437, 2004.

[3] M. Certicky, "Implementing a wall-in building placement in StarCraft with declarative programming," *arXiv preprint arXiv:1306.4460*, 2013.

[4] F. Richoux, A. Uriarte, and S. Ontañón, "Walling in strategy games via constraint optimization." in *AIIDE*, 2014, pp. 52–58.

[5] L. Perkins, "Terrain Analysis in Real-Time Strategy Games: An Integrated Approach to Choke Point Detection and Region Decomposition," in *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010*, G. M. Youngblood and V. Bulitko, Eds. The AAAI Press, 2010, pp. 168–173.