Wall-in

February 20, 2014

1 What we want

We want a pluggable module to decide how to make our wall, according to user's situation (position, chokes, ...) and desires (low cost, fast to build, no or few gaps for small units, ...).

Question: Is wall strength interesting? Because buildings composing a wall are more or less the sames, and a wall strength is more likely to correspond to the building with the less HP rather than the sum of buildings' HP

I don't think it is interesting. Since the most important thing is building a wall as soon as possible, moreover SCVs can repair buildings.... – *Alberto*

1.1 Library functionality

One easy and straight forward way to proceed is to propose a library with one function only:
Function **makeWall**
Inputs:

- a chokepoint (from BWTA::Chokepoint),
- a boolean telling if one wants a wall inside or outside the user's region, this is a little tricky... how you define inside/outside?? Maybe the best option is to give the BWTA::Region where you want to build the wall (remember that BWTA::Chokepoint has the info of what regions it connects) – *Alberto*
- what to optimize (cost, time, gaps). Start with one objective at the same time. Multi-objective optimization problems are quite tricky, time is relative, you need to send a SCV there and you can be interrupted by unpredicted events... on top of that you need to wait for resources or for building requirements constrains. In other words, some times you can build the wall in parallel other times in sequence... – *Alberto*
- an option for getting outputs as a list of tuples (building, position, timing) or as a list of BWAPI::UnitCommand. Mmmm I didn't know that we can create BWAPI::UnitCommand ... interesting ... – *Alberto*

Output: Depending of input options, a list of tuples (building, position, timing) or as a list of BWAPI::UnitCommand.

1.2 Inside makeWall

For BWTA, a chokepoint can have different aspects, and I think it is good to not limit ourselves to chokes linking low- and high-grounds only. See Fig. 1.

Let consider one wants to make an inside-wall from choke in Fig. 2.

Figure 1: Two different kind of chokes, circled in red.

Figure 2: Yeah, this one.

In order to make easier the optimization process, we could model the space around a choke through a 16×16 grid of tiles, like Michal proposed in his paper we can optimize this number using the width of the chokepoint – *Alberto*. See Fig. 3.

Then, we mark which cells correspond to unbuildable tiles, and marks also a source and target cell/tile. How we are going to "automatically" mark a source/target cell? – *Alberto* See Fig. 4. Remark that target cell let a "door" for an entrance/exit.

May be clearer if I remove the background. See Fig. 5.

Finally, knowing if the user wants to build a wall inside or outside his region, we can simplify the grid. Even if it is not shown in Fig. 6, we could just keep 4 tiles to the left/right of the starting tile and 3 tiles above/below the goal, since hugest buildings are 4×3 tiles big. I love this idea of simplify the grid, but we need to be careful to not simplify to much, sometimes we need to move far away from source/target cells to build the wall – *Alberto*

Figure 3: Choke abstracted via a grid.

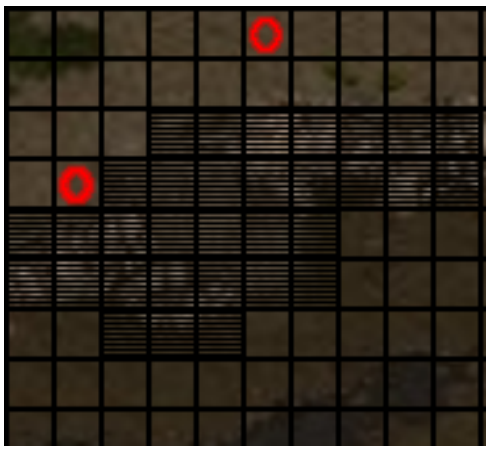

Figure 4: Remove unbuildable tiles and fix a starting and aimed tile.

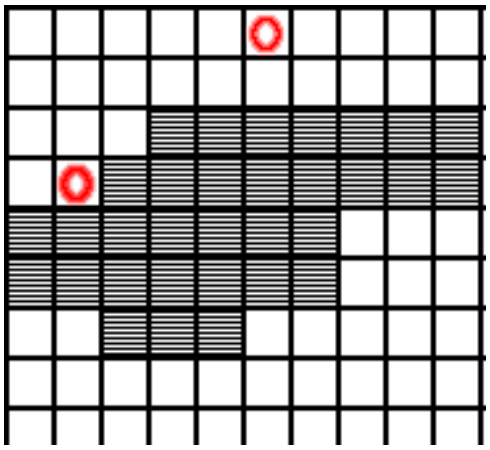

Figure 5: Better indeed.

2 CSP model

Let's focus an Terran first. A CSP is composed of a set V of variables, a domain D (i.e., values that variables can take) and a set C of constraints.

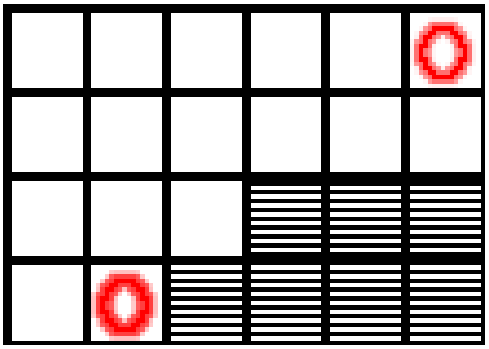

Figure 6: Simplify the grid.

2.1 Informal

Our **variables** are buildings. Thus, V could be composed of:

- an academy,
- barracks,
- an engineering bay,
- a turret,
- a factory,
- **two** bunkers,
- and **two** supply depots.

I think allowing two bunkers and supply depots extends greatly our possibilities (especially for having few gaps in the wall). Why twice these buildings and not the others? Good question; we could indeed double each buildings. we can start with this and experiment with adding more buildings (specially when we will test "long" chokepoints) – *Alberto*

The **domain** is the set of positions (i.e., cells) in the grid. A building position is characterized by its upper-left tile.

Constraints are:

- *Can I build here?* That is, if I place a given building to a given position, will it overlap some marked cells?
- *Non-overlapping buildings*, i.e., if I place a given building to a given position, will it overlap some cells booked for other buildings?
- *No tile-sized gaps between buildings*, that is, our wall must by composed of side-by-side placed buildings.
- If we want a "door" is it a constraint? – *Alberto*
- And if we want the spawn point of the Barracks? – *Alberto*

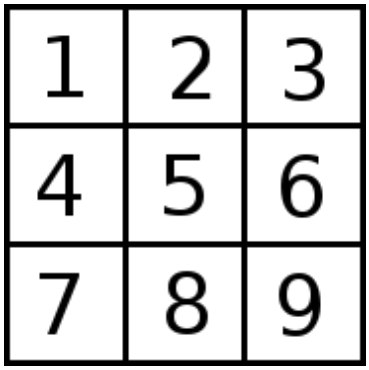

Figure 7: Positions on the grid.

2.2 More formal

2.2.1 Positions

Given a grid $n \times m$, let a position to be an integer from 1 to n*m, placed as shown by Fig. 7

Knowing the grid size, it is trivial to convert such a position into (x,y) coordinates. Thus in the following, I may use positions as an integer or as a tuple (x,y).

2.2.2 Variables and Domain

$$\begin{aligned} V &= \{A, B, E, T, F, U_1, U_2, S_1, S_2\} \\ D &= Positions \cup \{0\} \end{aligned}$$

In addition of a (changing) value, variables have also constant data, like *length* and *height* for the number of tiles needed to construct the corresponding building.

Position 0 means the building is not used in the wall.

Let c_{ij} be the cell at row i column j in the grid.

Let t_{ij} be a cell/tile filled by a building b, such that $t_{11} = b$ (recall: the value of b is the upper-left tile of the building), with $1 \leq i \leq b.length$ and $1 \leq j \leq b.height$.

2.2.3 Predicates

- A cell c_{ij} is buildable if the predicate $isBuildable(c_{ij})$ is true.
- A path from the source cell to the target cell exists if the predicate $path(c_{source}, c_{target})$ is true.

2.2.4 Constraints

Can I build here:

$$\forall b \in V, \forall t_{ij} s.t. t_{11} = b, 1 \leq i \leq b.length, 1 \leq j \leq b.height, isBuildable(t_{ij})$$

Non-overlapping buildings:

$$\begin{aligned} &\forall b_1, b_2 \in V, \\ &\forall t^1_{ij} \ s.t. \ t^1_{11} = b_1, 1 \leq i \leq b_1.length, 1 \leq j \leq b_1.height, \\ &\forall t^2_{i'j'} \ s.t. \ t^2_{11} = b_2, 1 \leq i' \leq b_2.length, 1 \leq j' \leq b_2.height, \\ &t^1_{ij} \neq t^2_{i'j'} \end{aligned}$$

No tile-sized gaps between buildings:

$$path(c_{source}, c_{target})$$

2.2.5 About the path

We could run a A*-like algorithm to determine if such a path exists.

If it does not exist, then we need to know how many tiles are missing to make a path (i.e., how many tiles are forming tile-sized gaps). This is necessary for making an objective function in order to allow efficient optimization of this constraint.

Knowing there is no path between the source and the target cell, we could make successive runs of an A*-like algorithm allowing to mark one cell, the two cells if we still have no path by adding one cell, then three cells and so on.

2.2.6 An alternative to *path*

Because running multiple times an A*-like algorithm is quite unsexy.

We can rewrite the *No tile-sized gaps between buildings* constraint as follow: a wall is sound if there exists two buildings having a building just on their side, and all other used buildings have two buildings on their side.

Thus, it means the two buildings having a building just on their side may not have this building both on their top, right, bottom or left, and all other used buildings have two buildings on their side must respect one configuration depicted in Fig. 8. Idea: can we incorporate non-walkable tiles in this formula? I'm thinking to check if the wall is "attached" to the *natural wall* :P – *Alberto*

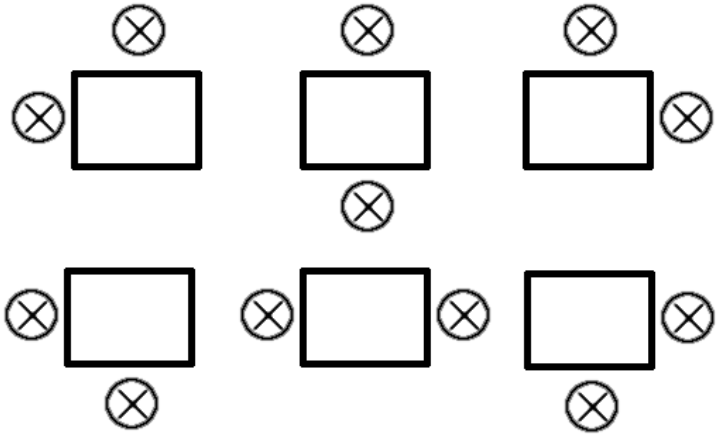

Figure 8: Possible configurations for buildings in the middle of a wall.

No tile-sized gaps between buildings:

$$
\begin{aligned}
&\exists b_s, b_e \in V,\ b_s \neq b_e,\\
&\Big(\exists b'_s, b'_e \in V,\ b_s \neq b'_s,\ b_e \neq b'_e,\ s.t.\\
&\neg\left(b_s.x + b_s.height + 1 = b'_s.x \wedge b_e.x + b_e.height + 1 = b'_e.x\right)\\
&\wedge\\
&\neg\left(b_s.x = b'_s.x + b'_s.height + 1 \wedge b_e.x = b'_e.x + b'_e.height + 1\right)\\
&\wedge\\
&\neg\left(b_s.y + b_s.length + 1 = b'_s.y \wedge b_e.y + b_e.length + 1 = b'_e.y\right)\\
&\wedge\\
&\neg\left(b_s.y = b'_s.y + b'_s.length + 1 \wedge b_e.y = b'_e.y + b'_e.length + 1\right)\Big)\\
&\wedge\\
&\Big(\forall b \in V,\ b_s \neq b \neq b_e,\ b \neq 0,\\
&\exists b_a, b_b \in V, b_a \neq b \neq b_b,\ s.t.\\
&(b.y = b_a.y + b_a.length + 1 \wedge b.x = b_b.x + b_b.height + 1)\\
&\vee\\
&(b.x = b_a.x + b_a.height + 1 \wedge b.x + b.height + 1 = b_b.x)\\
&\vee\\
&(b.x = b_a.x + b_a.height + 1 \wedge b.y + b.length + 1 = b_b.y)\\
&\vee\\
&(b.y = b_a.y + b_a.length + 1 \wedge b.x + b.height + 1 = b_b.x)\\
&\vee\\
&(b.y = b_a.y + b_a.length + 1 \wedge b.y + b.length + 1 = b_b.y)\\
&\vee\\
&(b.x + b.height + 1 = b_a.x \wedge b.y + b.length + 1 = b_b.y)\Big)
\end{aligned}
$$