ReignOfDKE
A bot for Battlecode 2014

F. Braun, A. Ludermann, A.J.J. Valkenberg

February 3, 2015

Abstract

This article presents the bot "ReignOfDKE". It was implemented for the MIT Battlecode competition 2014 and beats the reference player provided by MIT on 36 of 83 maps. The article describes the features and techniques implemented in this bot, including relevant topics like macro management, micro management and the used pathfinding techniques.

Disclaimer

This article was written for the research semester project of the Master "Artificial Intelligence", Department of Knowledge Engineering, Maastricht University. The project took place in winter semester 2014/15 half a year after the official Battlecode competition by MIT. Therefore the developed bot did not compete in this official tournament but only in the course internal tournament at the Department of Knowledge Engineering.

1 Introduction

The bot "ReignOfDKE" was implemented for the MIT Battlecode competition of 2014. This article starts by explaining the general macro strategy of the bot, how it is managed and executed. The subsequent section covers the micro management of single units, including amongst others the soldier's local combat behaviour, the Noise Tower behaviour and the different pathfinding techniques which are used in this bot. Afterwards, the used test framework will be presented, followed by the test results. To conclude the article, these results are discussed.

2 Macro management

This section includes the bot's general coordination, planning and construction of buildings and any other large scale decisions which have an impact on many units. First, the team management will be described which includes the building of different teams and the assignment of tasks. It also covers the decision making about which task will be assigned when and to which team.

2.1 Team management

The tasks that need to be fulfilled in Battlecode are best done by several soldiers together. An obvious situation is in fights against the opponent, be it to destroy its PASTRs or to defend the own. But also if PASTRs or Noise Towers should be built, several soldiers with the same task can protect each other if they encounter opponent's soldiers. This suggests the approach to divide the soldiers into teams.
Each soldier the HeadQuarter (HQ) produces, gets assigned to one of three teams of which each has a different main task.

- Team 0 is the flexible army team. Its main task is to attack the opponent and to destroy its PASTRs but it can also defend the own PASTRs if the opponent does not have one.
- Team 1 is the first builder team. After it builds a PASTR on a specific location and a Noise Tower next to it, the rest of the team defends them.
- Team 2 is the second builder team. After it builds a PASTR on another location and a Noise Tower next to it, the rest of the team defends them. In matches against the reference player this is a safe behaviour because it switches into a kamikaze mode as soon as his opponent has more than two PASTRs.

In an experiment which can be seen in section 5.1 some possible distributions between the different teams were tested. As the result, on small maps the teams will have the distribution 25%, 25%, 50% while on other maps the distribution 50%, 25%, 25% is maintained.

2.2 Tasks

Despite the required tasks like "build a PASTR" and "build a Noise Tower", three additional ones were defined.

- *BUILD PASTR:* The team moves to the given location using a complete pathfinding algorithm. The soldier to reach it first starts building the PASTR while the rest switches to the next task.
- *BUILD NOISE TOWER:* The team moves to the given location using a complete pathfinding algorithm. The soldier to reach it first starts building the Noise Tower while the rest switches to the next task.
- *GOTO:* Move to the given location using a complete pathfinding algorithm. This is used to reach specific locations like the opponent's PASTR.
- *ACCUMULATE:* Move towards the given location using a greedy pathfinding algorithm. The target needs not be reached, moving in its direction is enough. This is used to gather the teams in the beginning of the match.
- *CIRCULATE:* Move towards the given location using a complete pathfinding algorithm. If in a certain range around the target (between 10 and 30 squared distance), do random moves. If too close to the target move away from it using a greedy pathfinding algorithm. This task is used among others to surround own PASTRs which helps defending them and gathering more milk at the same time.

2.3 Task assignment

A centralised approach is used to assign the tasks. Each round, the HQ analyses the game state and coordinates the teams accordingly. If enough soldiers have been produced, it checks for the opponent's PASTRs. If there are any it sends out the army team to those locations, otherwise the army team is sent to defend the first own PASTR. If the opponent has more than 40% of the required milk a panic mode sets in and the other teams are sent to the opponent's PASTR locations as well to prevent losing. If the opponent has less milk, the other teams get the task to build a PASTR, each on a different location. If one or two PASTRs are already build, the respective team gets the task to circulate around the PASTR to defend it.
Despite the described approach, each soldier can also change its team's task. This adds flexibility and furthers a fast reaction to new events without waiting for the next round. For example this happens right before a soldier starts building something. If it is going to build a PASTR, it assigns the task of building a Noise Tower to its team. If it is going to build a Noise Tower, it assigns the task "CIRCULATE", which leads to the rest of the team defending the soon to be built Noise Tower.

2.4 Second calculation unit

To speed up the performance of the HQ one soldier is used for doing repetitive calculations in parallel to the HQ. This leads to the necessary byte code being distributed over two units. The soldier is referred to as "second core" whereas the HQ corresponds to the "first core". It is important to note that the second core is not invulnerable unlike the HQ, but during tests it turns out that it is not opposed to great danger as it stays close to the HQ and doesn't search for combat.

The second core is used for the following calculations:

- It determines the complexity of the map and broadcasts this information. Other soldiers decide which pathfinding algorithm to use based on this information (see also section 3.3).
- On startup it supports the other soldiers to create a simplified representation of the map (see also section 3.2.5).
- It determines promising locations for construction of PASTRs (see section 2.4.1).
- It analyses the behaviour of the opponent, more specifically it determines the average location of all broadcasting soldiers of the opponent. Additionally it senses the amount of milk, the opponent currently owns and broadcasts this information.

2.4.1 Good PASTR locations

Each 120 rounds the second core updates the most promising PASTR location for each team. It evaluates a subset of the tiles of the map regarding the following criteria:

- How high is the cow growth rate on this tile and its surrounding tiles?
- How far is the location from the opponents HQ?
- How far is the location from the opponents average location of broadcasting soldiers?
- Is the location too close to another promising PASTR location?
- Is the location too close to an existing PASTR?

The two latter criteria are strong boolean conditions. If a location is too close to an existing PASTR or another promising location it is not evaluated at all. Too close is defined as an overlap of the PASTR herding areas if a PASTR was built on each location.

The first three criteria result in a rating for a location: The product of the cow growth rate and the average distance to the locations to be avoided give the evaluation value. The location with the highest value is chosen as the new best PASTR location.

3 Micro management

The term micro management refers to the behaviour of single units. It includes their reaction to nearby events and their individual actions to reach the goals defined by macro management. This is reflected in the implementation of the bot. A Soldier repetitively executes the micro management, which is explained in depth in the next sections.

3.1 Local combat mode

In the beginning of executing the mirco management the soldier checks if there are soldiers of the opponent in range. If this is the case the soldier enters into a special behaviour mode which is referred to as "local combat". In the local combat mode the soldier has to act very fast as it has to flee from or attack the opponent's soldiers immediately. Therefore in this mode only a basic greedy pathfinding algorithm is used.To determine if the soldier should flee or attack the force of the local armies are compared. If the opponent's soldiers are dominating the soldier is going to flee, otherwise it is going to attack one of the opponent's soldiers in range.
In the local combat mode any other information such as the current task of the soldier (e.g. to build a PASTR) is ignored.

3.1.1 PASTR self-destruction

The PASTR self-destruction is closely linked to the local combat behaviour of the soldier. It is motivated by the milk bonus an opponent receives for destroying a PASTR. To keep the opponent from getting this bonus, the soldiers try to destroy their own PASTR when they are not able to protect it. When a PASTR is attacked it broadcasts its location to all soldiers. Now when the soldiers are close to the broadcasting PASTRs and they notice that they are not able to defend the PASTR they try to destroy it before the opponent does.

3.2 Pathfinding

Pathfinding is the most byte code consuming technique required in the game of Battlecode. It is of high interest to develop a pathfinding technique which produces good paths with high efficiency on all different map topologies.

In this section five different pathfinding algorithms are evaluated and the most appropriate one is determined. The comparison of the algorithms relies on three different maps of the Battlecode 2014 competition: "S1", "Rorschach" and "Unself" (MIT, 2010). Each incorporates different challenges and characteristics. The algorithms must find a valid path from one HQ to the other one. See Appendix A for the exact paths the different algorithms produce for map S1.

Each algorithm needs a certain time to initialise. This time is referred to as "initialisation time". After that the "moving time" start which refers to the time reaching the target after finishing the initialisation. The "moving time" represents the quality of the actual path.

3.2.1 Greedy

The "Greedy" algorithm tries to move towards the target by default. When it encounters an obstacle it performs three random moves.

3.2.2 Snail trail

The "Snail trail" algorithm tries to move towards the target by default. When it encounters an obstacle it tries a different direction. It only returns to a previously visited tile if there is no alternative.

3.2.3 M-Line-Bug

The "M-Line-Bug" algorithm determines a direct path to the target. It traverses around obstacles in that direct path until reaching it again. It is implemented based on the description of "Bug 2" by (Choset, Hager, & Dodds, 2010).

3.2.4 A*

The "A*" algorithm is the classical A*-algorithm from literature. Here a weighted version is used. The admissibility criterion is violated to increase the performance of the algorithm. The overestimation factor of the heuristic score is 3. This converts the algorithm in a best-first-search-variant of A*.

3.2.5 Divide&Conquer-A*

To increase the performance of the classical A* here the algorithm is executed on a simplified map which never exceeds the boundaries 20x20. For each square of this reduced map the algorithm determines if the square is traversable based on the tiles that it contains. A square is treated as traversable when it contains at least as many traversable tiles as blocked tiles. The reduced map is broadcasted to be accessible by all soldiers. This reduces the initialisation time of a Soldier as it can reuse the knowledge which was already acquired by other soldiers. To navigate from one square to the next the normal A*-algorithm is used. This is a safe choice. Tests with the Snail trail algorithm as the fine granular algorithm failed, because the simplified map is not accurate enough. Sometimes a connection between to squares does not exist, although it does according to the simplified map. In these cases A* is a safe fallback.

3.3 Evaluation of pathfinding algorithms

Because of the evaluation results the pathfinding was implemented according to the following rules: When it is not important that a target is reached, the Greedy algorithm is used. This is the case for example, when a soldier has to flee

algorithm	Greedy	Snail-trail	M-Line-Bug	A*	Divide&Conquer A*
initialisation time	near zero independent from map	near zero independent from map	short but depends on distance to target	very long and dependent on distance to target, sometimes > 100 rounds	about 25 rounds independent from map size
completeness	not complete	complete (for prove see Appendix)	complete	complete	complete
movement time / path quality	low	low	low	good	satisfactory
worst case scenarios	gets stuck easily	greedy behaviour leads to dead ends getting fully traversed	sometimes traverses around the entire map border	initialisation time is not acceptable (soldier freezes during this time)	map simplification can lead to bad paths but still acceptable

Table 1: Pro and Contra of pathfinding algorithms

from opponents. The Snail trail and M-Line-Bug algorithms produce good paths regarding their initialisation time. Unfortunately both have unacceptable worse case scenarios. A* produces good paths but has an unacceptable initialisation time. Therefore the final implementation uses A*-Divide&Conquer when the target has to be reached. This algorithm is a good trade-off between accuracy and efficiency. Furthermore it has the huge advantage that the soldiers do not freeze for a large amount of rounds. Hence the soldiers are able to react to approaching opponents or changes to their assigned tasks.

There is one Exception to choosing A*-Divide&Conquer: When the map contains few or no obstacles, the M-Line-Bug algorithm is used instead of A*-Divide&Conquer. The complexity criterion is based on the tiles that are located between the upper left corner and the middle of the map. If any tile on this line is of terrain "VOID", the map is estimated as complex and A*-Divide&Conquer is used. In the reverse case M-Line-Bug is used, which leads to a large boost in performance.

3.4 Noise Tower behaviour

The Noise Tower behaviour is based on the fact that a Noise Tower is always constructed adjacent to a PASTR. Subsequently a Noise Tower should aim to pull the cows towards itself and indirectly into the range of the nearby PASTR. The Noise Tower shoots in a certain direction starting with the maximum shooting range. Subsequently it decreases the range step by step until it is smaller than the PASTR's herding range. Then the Noise Tower rotates the shooting direction and starts over pulling cows again. The sequence of the directions is illustrated in figure 1.

3.5 Leadership

In an attempt to reduce the amount of time soldiers stood still to calculate their path, a leader functionality was created. Each team of soldiers had one leader which they followed. This would allow the soldiers to remain active and close to the leader while it took the time to calculate the best path.

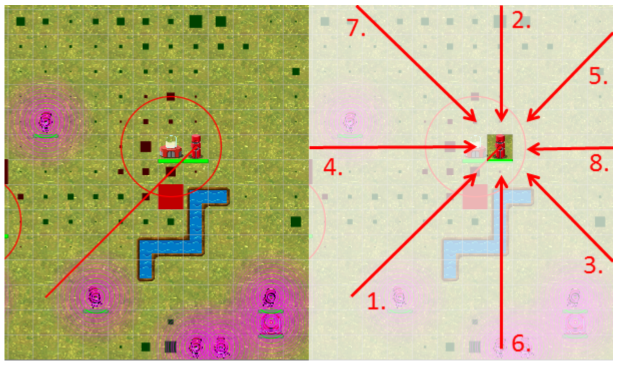


Figure 1: Noise Tower behaviour. The left side shows a sample game situation and the right side shows the resulting sequence of directions.

3.5.1 Implementation

The leader functionality was only active when the "GOTO" task was assigned to a team. Of that team, the soldier with the lowest ID was assigned to be the leader. That ID was broadcast to one of the team's channels, so each soldier could check if they were the leader of their team. If a soldier found itself to be the leader, it would start to calculate a path to the target of the "GOTO" task, using the A* pathfinding method as described in section 3.2.5. When moving along the calculated path, the leader soldier set its current location as a temporary target, which was broadcast into the team's channel. If a soldier was not the leader, it checked if a temporary target was set and started to calculate a path to the temporary target, also using A*. An extra feature was implemented that allowed the leader to wait for a fraction of its team-members to be close before it would make a move. This feature was added to avoid the leader setting temporary targets which were too far away from its team-members to quickly calculate a path to, which would defeat the purpose of the leader.

3.5.2 Testing

In order to check how the implemented leader behaviour affected the performance, several tests were run versus a previous version of the "ReignOfDKE" player. A match was run on 9 maps, with each player being player A and B, resulting in 18 matches in total. A version of the same code, but without the leadership feature, was also tested in the same way. This enables a clear performance comparison between the code with and without the leadership feature. The results of these matches can be seen in table 2.

These results indicate that the leadership feature did not increase the performance. It actually caused a decrease in performance since it won less matches, and the matches that were won took on average 300 rounds longer than without the leadership feature. After reviewing the replay files of the test matches, several reasons for the decrease in performance became apparent.

- The waiting behaviour caused the movement of the team to slow down. With small team sizes, a newly spawned team member is likely to cause the leader to have to wait.

- The leader being killed causes one of the other team members to consider itself the new leader. While this is intended behaviour, it will cause the soldier to have to recalculate its path, since it is now no longer supposed to go to the temporary target, but the original target.

	Matches won	Matches lost	Average rounds to win	Average rounds to lose
Leadership feature	5	13	1747	1431
Without leadership	10	8	1437	1331

Table 2: Leadership feature test results on 9 maps over 18 matches.

4 Testing

4.1 Framework

To reduce the time consumed by testing, a framework that automatically runs Battlecode matches was created. While it is possible to run a match between two Players on multiple maps, for testing purposes the graphical interface is not required. The implementation of this framework is based on a Python script provided by the Battlecode course of 2014. However, the authors decided to code the framework in Java, due to limited knowledge of the Python language. For testing purposes, the ”file” target in the Battlecode Ant build file was chosen. This runs a single match of Battlecode and saves it into a file, which is essentially a replay file. By adjusting Battlecode's configuration file, the players and the map for which the match is run can be defined. Executing Ant inside of the framework code allowed the output of the Battlecode client to be captured and processed. By parsing the output using a regular expression, the winning player can be determined. Repeating this process allows testing of a new implementation or feature on many different maps, after which the results can be analysed.

4.2 Methodology

While the maps provided by the Battlecode environment are mirrored to ensure a fair and equal starting position for both players, bugs or heuristics in pathfinding algorithms require tests to be run from both starting positions. The Battlecode course at MIT defined three sets of three maps on which to beat their provided ”reference player”. These maps are the main focus for tests and are referred to as the ”reference maps”. It is feasible to expect that certain strategies can perform better or worse based on the size of a map. Therefore the maps available in the Battlecode environment were classified into three types.

4.2.1 Map classification

Each map was classified to be either of "SMALL", "MEDIUM" or "LARGE" size. To determine a map's type, the following rules were applied:

- If both dimensions are smaller than 40, the map is "SMALL"
- If both dimensions are smaller than 60, the map is "MEDIUM"
- If both dimensions are larger or equal to 60, the map is "LARGE"

If none of the rules above apply, the map has unequal dimensions. In such a case the map's type is defined by it's largest dimension, and the following rules are applied:

- If the largest dimension is smaller than 50, the map is 'MEDIUM'
- If the largest dimension is greater than or equal to 50, the map is 'LARGE'

See Appendix C for a full list of the maps available in the Battlecode environment and their classification. 83 out of 88 maps have been classified. The 5 maps that were not classified have been removed from the testing sets due to either having no reachable locations with cow growth, being intended for navigation testing purposes, or a significant alteration of game rules (i.e. decreasing milk quantities).

5 Results

5.1 Team distribution

Table 3 presents the results of the team distribution tests. The following distributions were tested:

- Distribution 1: Team 0 50%, Team 1 25%, Team 2 25%
- Distribution 2: Team 0 25%, Team 1 50%, Team 2 25%
- Distribution 3: Team 0 25%, Team 1 25%, Team 2 50%

These distributions were tested against the reference player provided by MIT and against the other distributions, from both starting positions. The tests were performed on the set of reference maps, and on the SMALL, MEDIUM and LARGE map sets.

The overall performance of the distributions is comparable. However, it is clear that distribution 3 performs worse on the reference maps; 6 wins on average versus 10 and 9.5 on average for distribution 1 and 2 respectively. One more difference is that distribution 3 performs better on SMALL maps and worse on LARGE maps versus distribution 1 and 2.

Distribution 1	vs. reference player	vs. Distribution 2	vs. Distribution 3
Reference maps	8	9	13
SMALL maps	18	25	21
MEDIUM maps	28	35	36
LARGE maps	22	23	26
Distribution 2	**vs. reference player**	**vs. Distribution 1**	**vs. Distribution 3**
Reference maps	8	9	12
SMALL maps	18	25	20
MEDIUM maps	28	35	36
LARGE maps	22	23	26
Distribution 3	**vs. reference player**	**vs. Distribution 1**	**vs. Distribution 2**
Reference maps	7	5	6
SMALL maps	18	29	30
MEDIUM maps	30	34	34
LARGE maps	20	20	20

Table 3: Amount of matches won.

5.2 Reference player

Table 4 shows the results of the tests against the MIT reference player. The tests were performed on all 83 test maps, which include the reference maps, from both starting locations.

These results show that the ”ReignOfDKE” player performs worse on SMALL maps than on other map types against the reference player. However, on the reference maps it has an equal win rate.

	Matches won	Matches lost
Reference maps	9	9
SMALL maps	16	32
MEDIUM maps	35	35
LARGE maps	21	25

Table 4: Performance of ReignOfDKE versus MIT reference player.

6 Conclusion

The bot ”ReignOfDKE” wins on average 36 out of 83 maps against the reference player.

During the creation of test statistics it was noted that using different devices has an impact on the results. Some of the before lost maps changed to wins and vice versa when using another computer. The exact results can be found in Appendix D.

On many of the maps the match is very close, the winner only earns a small amount of milk more than the opponent. Generally the strategy of building two PASTRs and Noise Towers turns out to be effective against the reference player. The lost matches mostly correspond to matches where building two PASTRs fails for different reasons.

The construction of four buildings lead to a huge disadvantage in army size. Therefore a more aggressive opponent than the reference player might easily win against "ReignOfDKE". To face this issue it would be necessary to build an opponent model and adapt the macro strategy according to it.

"ReignOfDKE"'s pathfinding is more advanced than the pathfinding of the reference player. Therefore ReignOfDKE tends to perform better on maze-like maps. Opposed to that the advanced pathfinding generally requires more time both in its initialisation and during the movement. Therefore the reference player shows a more dynamic behaviour and reacts faster to changes.

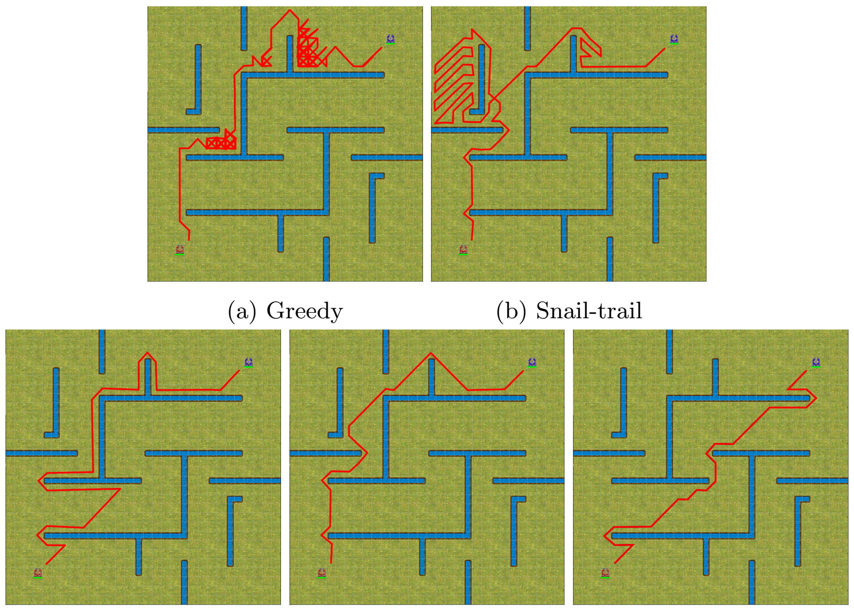
References

Choset, H., Hager, G., & Dodds, Z. (2010, August). *Robotic motion planning: Bug algorithms.* Retrieved from `\url{http://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf}`

MIT. (2010). *Battlecode.* https://www.battlecode.org.

7 Appendices

7.1 Appendix A - pathfinding analysis



(a) Greedy (b) Snail-trail

(c) M-Line-Bug (d) A* (e) A* Divide & Conquer

Figure 2: Each algorithm's path on map S1

	Rounds needed for path calculation	Number of Moves	Rounds needed for movement
Greedy	0,0098	471	783,4122
Snail-trail	0,0131	103	246,4050
Bugging	0,9310	62	130,4822
Weighted A*	70,5087	36	88,9134
Divide&Conquer A*	19.8042	37	147,6593

Table 5: Individual algorithm performance on map S1

	Rounds needed for path calculation	Number of Moves	Rounds needed for movement
Greedy	0,0098	∞	∞
Snail-trail	0,0131	152	351,4207
Bugging	2,1834	94	182,2466
A*	335,1310	60	129,3079
Divide&Conquer A*	34,1617	70	241,9587

Table 6: Individual algorithm performance on map Rorschach

	Rounds needed for path calculation	Number of Moves	Rounds needed for movement
Greedy	0,0599	182	342,2337
Snail-trail	0,0131	87	202,2755
Bugging	3,5055	98	219,7794
A*	211,9317	86	195,3621
Divide&Conquer A*	34,6312	90	282,2732

Table 7: Individual algorithm performance on map Unself

7.2 Appendix B - Snail trail algorithm completeness prove

This is an informal prove why the Snail trail pathfinding algorithm is complete. A pathfinding algorithm is complete if and only if it always find an existing path between two locations as long as one exists. Resulting we can assume for the prove that a path exists between initial location and target.

The Snail trail behaviour can be divided into three cases:

1. From the current location the unit can move to the next location on direct path to the target.

2. From the current location the unit can *not* move to the next location on direct path to the target. There are unvisited neighbours.

3. From the current location the unit can *not* move to the next location on direct path to the target. All neighbour location have been visited already.

As long as the first case applies the prove is trivial. The unit will just move on until reaching the target. In the second case the Snail trail algorithm will always choose a neighbour it has not visited before. So it will always traverse all tiles contained in a dead end. I the third case it will always choose the location it has not visited for the largest amount of time. This is the key for the completeness of the algorithm. It will always move backward on the already traversed path choosing the tile that has been visited longest ago. It only stops doing so if there is a neighbour it has not visited before (case 2). Consequentially in the worst case all reachable tiles on the map will be visited. If the target is reachable, then it is contained in this set of reachable tiles and therefore it has to be visited at a certain point. Therefore the Snail trail algorithm is complete.

7.3 Appendix C - Battlecode maps

Maps that were not classified: battlefield2, cow, navtest, omgrusrs and ww.

Name	Type	Name	Type
acht	SMALL	manhattan	MEDIUM
almsman	SMALL	maseeh	LARGE
apartments	SMALL	meander	LARGE
ascent	MEDIUM	moba	SMALL
babble	SMALL	money	MEDIUM
backdoor	SMALL	moo	MEDIUM
bakedpotato	MEDIUM	neighbors	LARGE
blocky	MEDIUM	oasis	MEDIUM
boston	LARGE	offices	MEDIUM
cadmic	LARGE	onramp	SMALL
castles	SMALL	overcast	LARGE
clear	SMALL	pipes	MEDIUM
corners	SMALL	quadrants	MEDIUM
cubes	MEDIUM	race	LARGE
desolation	SMALL	reticle	SMALL
divide	SMALL	rorschach	LARGE
donut	SMALL	rushlane	LARGE
effervescent	MEDIUM	s1	SMALL
egg	MEDIUM	siege	LARGE
emoticon	MEDIUM	skinny	MEDIUM
fenced	MEDIUM	smiles	MEDIUM
fengshui	LARGE	spider	LARGE
filling	SMALL	spots	SMALL
flagsoftheworld	SMALL	spyglass	LARGE
flowerthing	MEDIUM	steamedbuns	MEDIUM
flytrap	LARGE	stitch	SMALL
friendly	MEDIUM	supersweetspot	MEDIUM
fuzzy	MEDIUM	sweetspot	MEDIUM
gilgamesh	SMALL	swirl	SMALL
highschool	SMALL	temple	MEDIUM
highway	LARGE	terra	MEDIUM
house	MEDIUM	territory	LARGE
hydratropic	SMALL	thermopylae	MEDIUM
hyperfine	MEDIUM	tortoise	MEDIUM
ide	MEDIUM	traffic	MEDIUM
intermeningeal	MEDIUM	troll	MEDIUM
itsatrap	LARGE	turtlemap	LARGE
jaws	LARGE	unself	LARGE
latch	SMALL	valve	SMALL
librarious	LARGE	ventilation	LARGE
longetylong128	LARGE	willow	MEDIUM
magnetism	MEDIUM		

Table 8: Map size classification

7.4 Appendix D - ReignOfDKE vs. MIT reference player final tests Battlecode screen-shots

not included in this version due to size.