Goal: develop an application that provides detailed analysis of optimal build strategies and allocation of resources, in relation to a game (http://leagueoflegends.wikia.com).

Background

The game can be divided into two main stages – early-game and end-game, with an extended transitional period between them. Our interest lies mainly in the first stage, where the quantitative aspects of our build matter the most. The second stage is more reliant on strategy, maneuvering, and objective control.

The first half of a match is spent 1v1 or 2v2 against an opponent in a lane. There are 5 players per team, 3 lanes, with 1 player, per team, roaming the map ambushing people. In each lane, the player has a defensive tower/turret that he can fall back to. AI minions spawn from each player's lane and battle each other in the contested area between towers. Scoring the killing blow on an enemy minion grants you a certain amount of gold, and is your primary source of income throughout the game. Last hitting a minion involves a certain amount of risk: your enemy is standing nearby and will opportunistically harass you. Dying to your opponent gives them a significant gold and experience advantage.

Through purchasing and upgrading equipment, gold is the primary means of augmenting our effectiveness. If we spend it unwisely, our opponent may attain dominance over the principal means of income. Once denied access to resources, a snowball effect ensues, leading us to fall further and further behind. Given the simplistic nature of the game, the choice of tactics and strategy is limited. Therefore, in relation to a user defined strategy (war of attrition, blitzkrieg, etc), we would like to define strong metrics that describe to us, in optimal terms, what to build, when and how to build it, and to develop an intuitive understanding of the utility gained by doing so.

Sections

1. End-game allocation of inventory slots

Treating this as a combinatorial problem: there are a finite amount of items to choose from – ~200, which can be further filtered down to ~50, based on context – and our player can hold only 5 at any given time. The number of combinations (with repetition of elements) is given by:

$$fcr(n,k)=\frac{(n+k-1)!}{k!(n-1)!}$$

$$fcr(50,5)=3162510,\ fcr(200,5)=2.80235004e+9$$

Unranking combinations

"A ranking algorithm determines the position (or rank) of a combinatorial object among all the objects (with respect to a given order); an unranking algorithm finds the object having a specified rank. Thus, ranking and unranking can be considered as inverse operations."[1]

Using the combinatorial number system, the N-th combination can be quickly unranked and measured, in parallel, by commodity graphics processing hardware (GPU).

$$set(n)=\{\{n,fcr(n,k)\},\{n-1,fcr(n-1,k)\},\ldots,\{0,fcr(0,k)\}\}$$

$$f(i,n,k)=Max\{x\in set(n),x_2\le i\}$$

$$unrank(rank,n,0)=\emptyset$$
$$unrank(rank,n,k)=\{x_1\}\cup unrank(rank-x_2,n,k-1)\ where\ x=f(rank,n,k)$$

Where the union operation appends the element to a sequence.

After applying our metric, and exhaustively searching all combinations through brute force, we have a good idea of what the optimal usage of inventory slots will look like for a final end-game build.

Notes

Initially, we tried to solve this combinatorial problem using integer programming and Mathematica's optimization routines. Due to our objective function involving a non-linear DPS metric, and a large amount of variables, we found this approach to be significantly slower than exhaustive search on the GPU. The GPU/OpenCL version completes in a matter of seconds, vs hours/days for the alternative. Perhaps this is symptomatic of the author's inexperience with formulating mathematical optimization problems.

1 D. L. Kreher and D. R. Stinson, Combinatorial Algorithms, Generation, Enumeration and Search (CAGES), CRC Press (1999), Chapter 2, "Generating Elementary Combinatorial Objects" pp. 31--32

2. Optimal build path

With our end-game configuration defined, we can begin to think about the optimal steps to build it. There are two types of items, organized into three tiers:

i. Basic/irreducible.
ii. Composite.
iii. Top level composite (can no longer be upgraded).

Each composite item has a dependency graph of items required to build it. This digraph is equivalent to a partially ordered set (poset).

A linear extension of a poset is a total ordering, which does not violate any of the relationships defined by the partial order, and is usually non-unique. The set of all linear extensions corresponds directly to the possible ways of constructing the item in question.

> ***Problem1:*** *Suppose we have a composite item **B**, that requires two of the same basic item **A** to construct, and an item **C** that requires **B**, and a third **A**. These types of build hierarchies don't function properly when treated as a regular poset: {**A** < **B**, **B** < **C**, **A** < **C**}, since the multiple quantities of **A** are lost.*
>
> *The simplest solution is to construct the graph making each **A** unique: A_1, A_2, A_3, but this generates superfluous linear extensions. The build order {A_1,A_3,A_2,**B**,**C**} is no different than {A_3,A_1,A_2,**B**,**C**}. These redundant extensions are sorely unwanted, considering how quickly the problem space expands when you merge two graphs that share no common nodes.*

Parallel generation of linear extensions

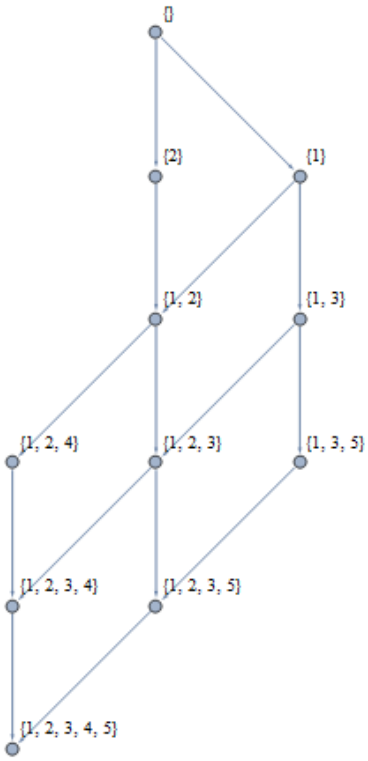
Given our earlier interest in using commodity graphics hardware to accelerate our computations, and due to the nature of combinatorial explosion, a parallel algorithm for generating linear extensions is desirable. At a cursory glance, most of the literature devoted to the subject of topological sorting seems to concentrate on sequential algorithms, which manipulate the output of the previous iteration through lexical shifts and rotations. Instead, we opt to adapt the approach of generating linear extensions via an intermediate lattice of ideals[2]. Originally used for generating extensions at random, the nature of the lattice representation transforms the problem into finding all max-length paths, from source to sink, in a graph.

---

2 "Efficient computation of rank probabilities in posets", Karel de Loof

Lattice of Ideals

"A downset or ideal of a poset (P,≤) is a subset D ⊆ P such that x ∈ D,y ∈ P and y ≤ x implies y ∈ D. If we consider the set of ideals I(P) and equip it with the set inclusion ⊆, which is a partial order relation, a new poset (I(P),⊆) is obtained. Moreover, it is more than a poset: it is a distributive lattice."3



For the poset $P=\{2<4, 1<3, 1<4, 3<5\}$ we obtain the above digraph and following lattice.

A linear extension of P corresponds to a max length path from $Source\{\}$ to $Sink\{1,2,3,4,5\}$, where traversing each node records the difference between parent and child. For example: traveling down the middle path we obtain the linear extension $L=2,1,3,5,4$.

3 Section: 2.2 "The lattice of ideals representation of a poset", "Efficient computation of rank probabilities in posets", Karel de Loof

Unranking linear extensions

Mirroring our earlier approach, we look for an efficient means of ordering and unranking our linear extensions.

"Suppose without loss of generality that there is one source $\emptyset$ *and one sink* S *. For each node* v *in your digraph, you can compute recursively the number* N_v *of unique maximum length paths from* $\emptyset$ *to* v *: at* $\emptyset$ *we have* $N_\emptyset = 1$ *, and at other points,* $N_v = \sum_u N_u$ *, where the sum goes over all immediate predecessors of* u *.*

This suggests the following encoding for maximal paths at a node v *. Suppose the immediate predecessors of* v *are* $u_1, \ldots, u_k$ *in some arbitrary but fixed order. We will encode all paths as numbers in the range* $[0, N_v) = \{0, \ldots, N_v - 1\}$ *in the following way: all paths through* u_1 *will be encoded in* $[0, N_{u1})$ *, all paths through* u_2 *will be encoded in* $[N_{u1}, N_{u1} + N_{u2})$ *, and so on. Inside each interval, we apply the same encoding scheme recursively. By applying this scheme for* $v = S$ *(the sink), we get an efficient encoding system for all maximal paths."4*

Greedy algorithms

"A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage5. Typically, a greedy algorithm is used to solve a problem with optimal substructure if it can be proved by induction that this is optimal at each step."

If our metric function permits such a heuristic, the search space can be drastically reduced by allowing for a single pass to be performed instead of millions.

4 Unranking paths in a graph/lattice (Yuval Filmus), http://cs.stackexchange.com/a/16437/10979

5 Introduction to Algorithms (Cormen, Leiserson, and Rivest) 1990, Chapter 17 "Greedy Algorithms" p. 329.

3. Metric functions

There are two primary tactics for building an advantage: “all-in” aggression and sustain based attrition. For the former, you pick a character with strong early-game dominance and build for winning duels – this usually means building Health and AD/AP. Knowing that you will win a prolonged exchange, you push your advantage and force the opponent into playing passively. Under the attrition strategy, you build sustain: life steal, health/mana regen, armor/magic resist, and disengage from any prolonged exchanges, focusing on trading blows optimally and outlasting your opponent.

Primary Goals

- Quantify the utility of each stat, in relation to the above mentioned strategies (or others, if you can think of them), and their change in value as time progresses. In the early levels, for example, intuition favors Health and AD/AP for duels, due to their additive nature and low starting values.
- Analyze the properties of our metric functions, determining their suitability for use in the previously mentioned greedy algorithm, and in general, determine if locally optimal == globally optimal (pseudo-convex function?).
- Discover the extent to which we can avoid comparative metrics and simulations. These both require a reference point, your opponent, which adds some ambiguity. Although, most players tend to follow very predictable and well known builds. Perhaps things are so context sensitive that there is no other way, but it would nice to operate in more abstract/general terms.

Secondary Goals

- When rating ability damage, consider separate ratings of burst damage, as well as sustained damage over time. A character's full combo may kill a weaker person from full health, but they won't necessarily have good sustained damage. Both aspects have different advantages, considering the time window to kill a person is usually fairly small, and that they will try to run away from you. For attrition strategies consider the mana cost of abilities over time.
- Discover best way to measure the utility of defensive stats without a reference point for incoming damage being dealt.
- Deal with multi-objective optimization.
- Deal with the advantages of having a ranged auto-attack vs melee.
- Analyze the relationship/tradeoffs between offense and defense. Consider a “peel” factor – peel being the amount of time in seconds you can rely on your teammates to keep an enemy from attacking you. Possibly look into how the skirmish size changes things: 1V1, 2v2, and 5v5 are the most common scenarios, yet in all situations its unlikely that you will have more than 2-3 people attacking you at any given time.
- Come up with a good way to account for damage from abilities. In practical terms, this is probably the most problematic. The way a character's abilities work is fairly nuanced, and hard to automatically mine from the game's data set. However an ability's AD and AP bonus damage scaling are easy to extract, and the flat base damages could just be factored out, since they aren't affected by gear anyway. This approach doesn't deal with abilities that enhance a

character's attack speed, or add on-hit effects to their auto-attacks though. If ability damage is rated/averaged in terms of damage per second, do we completely lose all feeling for the level of burst damage they achieve?

- Develop some optimal guidelines for how to gracefully lose vs an opponent that you can no longer contest. Do you build pure defense and hide under your tower? Do you build passive gold generation items to make up for your reduced capacity to farm?

Preliminary attempts at measuring utility

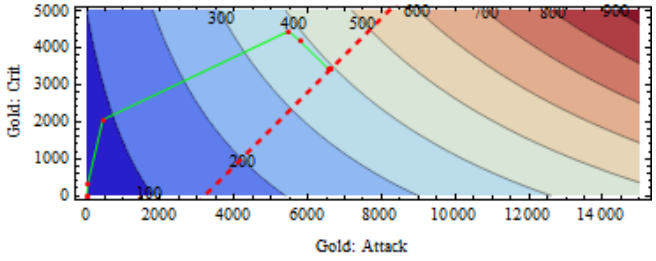
Simplified auto-attack DPS formula:

$$dps(goldAD, goldCrit) = (50 + (goldAD/36)) * (1 + (goldCrit/5000))$$

Where 1 attack bonus costs 36 gold, and 1% critical hit chance costs 50 gold. (we can ignore attack speed and armor penetration to simplify the discussion).

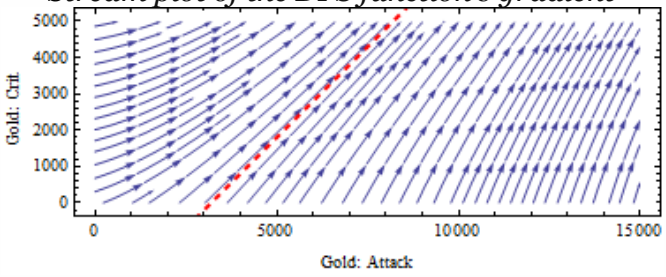
Initially, it makes sense to spend your gold on attack damage, since it increases additively. To double your initial DPS of 50/s you could spend 1750 gold on attack damage, or 5000 gold on critical chance. When you have reached 250 attack damage, it would cost 8750 gold to reach 500 DPS if you were to continue buying attack damage, or 5000 gold if you were to invest in critical hit chance. So, the utility of attack damage in relation to crit has decreased.

Contour plot of DPS based on gold spent



The green line is the path taken by an optimization function as it tries to find the locally optimal point at 10,000 gold spent: {ad -> 6600, crit -> 3400}. And the red line is the path of steepest ascent.

Stream plot of the DPS function's gradient



Path of steepest ascent

$$goldCrit = -5000 + \sqrt{(1800 + goldAD)^2}$$

Finding the 2D path of steepest ascent on a 3D surface

"For the curve y(x) to be tangent to ∇f, its slope must equal the rise-over-run of the 2d gradient vector."

$$f(x,y)=(50+(x/36))\cdot(1+(y/5000))$$

$$g=\nabla f=\{1/36(1+y/5000),(50+x/36)/5000\}$$

Solving the differential equation:

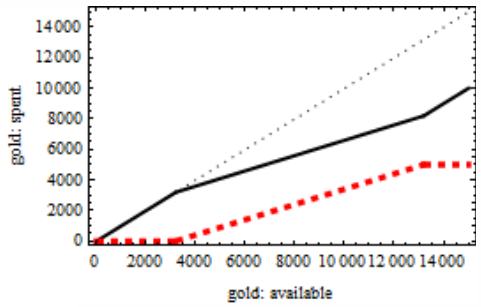
$$y'(x)=(g_2/g_1 where: y\to y(x)), y(6600)=3400$$

Gives:

$$y(x)=-5000+\sqrt{(1800+x)^2}$$

Visually, the line seems to make sense, and the values seem to correspond with those generated by Mathematica's optimization routines (they both cross zero at 3200 ad).

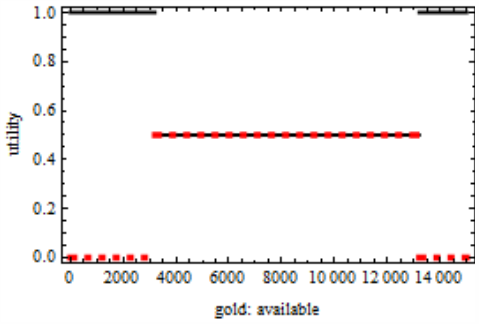
Now, using this relationship we can graph the amount spent per stat, relative to the total amount of gold available.



- Dashed red line: crit.
- Thick black line: attack.
- Thin dotted line: max spendable gold.

Taking the derivative of each seems to gives us a working concept of the stat's level of utility.

Assuming this is all correct: all gold should be invested in Attack until 3200 is spent, after which, gold should be evenly divided until critical hit chance maxes out at 100%. This seems to correspond with our earlier intuition.



These "utility" values indicate how the stats relate to the optimum distribution of gold to maximize DPS, but is this the measure of their true utility? Lowering the cost of a stat lowers its "utility" value (since less gold has to be allocated to it), which seems counter-intuitive.

Finding the 3D path of steepest ascent on a 4D surface

$$f(x,y,z)=(50+(x/36))\cdot(1+(y/5000))\cdot(1+(z/3333))$$
$$g=\nabla f$$

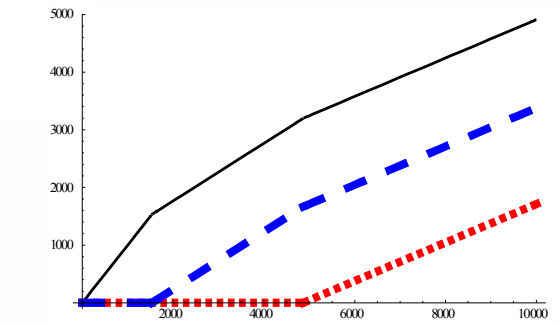
Solving: $y'(x)=g_1/g_2 where: y\to y(x),\ z'(x)=g_1/g_3 where:\{y\to y(x), z\to z(x)\},\ y(4911)=1711, z(4911)=3378$ Gives: $y(x)=-3200+x,\ z(x)=-1533+x$

Parameterize over gold:
$$Max(0,y(x))+Max(0,z(x))+Max(0,x)=gold,\ gold\geq 0$$

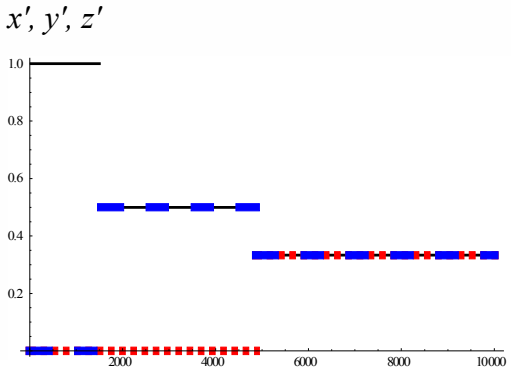
Gives us:

$$ad(gold)=\begin{cases} gold & if: gold\leq 1533 \\ \frac{1533+gold}{2} & if: gold\leq 4867 \\ \frac{4733+gold}{3} & if: gold>4867 \end{cases}$$

And the path of steepest ascent:
$$path(gold)=\{x, y(x), z(x)\} where\ x=ad(gold)$$



- black: ad(gold)
- blue wide dash: z(ad(gold))
- red small dash: y(ad(gold))

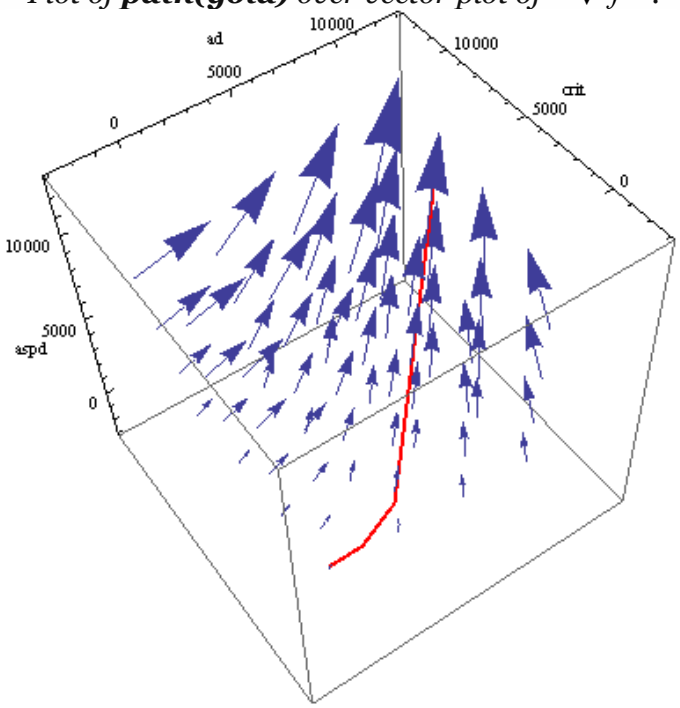


Comparison:

gold	Mathematica's FindMaximum	path(gold)
4867	{3199.81,0.373476,1666.81}	{3200,0,1667}
1000	{1000,0.0000382154,0.00030284}	{1000,0,0}
2000	{1766.5,0.0000343718,233.5}	{1766.5,0,233.5}

The path makes sense relative to the vector plot of ∇f :

Plot of ***path(gold)*** *over vector plot of* ∇f .



4. Game Background

- The player can choose from a variety of different characters with unique abilities, strengths and weaknesses.

- As the character gains EXP and levels up during a match, his base stats increase, and a point is awarded to be spent upgrading abilities. The level cap is 18. Gold and EXP are only earned when in contested areas.

- Each character has three regular abilities (upgradable 5x), one 'ultimate' (upgradable 3x) , and one passive. Ultimate ability upgrades are available at Level 6, 11, 16.

- Levels 1, 2, 3 and 6 may have extra kill potential, due to abilities being unlocked, depending on the match up.

- Abilities perform a variety of functions, but mostly they just deal damage. As an Example:
  Bladesurge:
  Range: 650,
  Cooldown: 14 / 12 / 10 / 8 / 6 Seconds,
  Cost: 60 / 65 / 70 / 75 / 80 Mana,
  Physical Damage: (20 / 50 / 80 / 110 / 140) +100% AD

  The fields with slashes are values based on what level the ability has been upgraded to. AD is the value of the character's Attack Damage stat.

- Characters have an auto-attack function. You click a target, and it automatically keeps dealing damage to it, based on certain offensive stats. Using an ability delays the auto attack cycle until the action completes.

- Characters can be grouped into three main archetypes. The first two require careful positioning, and support from teammates to function properly, due to lack of defense:

  - Archer: sustained damage: focuses primarily on increasing auto-attack damage.
  - Mage: burst damage: tries to maximize damage dealt with abilities.
  - Bruiser: balances offense/defense

- Offensive stats: Attack Damage (AD), Critical Hit Chance, Attack Speed, Ability Power (AP), Armor Penetration, Magic Penetration, Cooldown Reduction.
- Defensive stats: Health, Armor, Magic Resist.
- Sustain stats: Lifesteal, Spell Vamp, Health Regen, Mana, Mana Regen.
- Utility stats: Movement speed, Gold bonus.
- Examples of archetypical characters: Annie (mage, burst damage), Riven/Renekton (fighter, early-game bully), Ashe (archer, auto-attack based, abilities provide utility instead of damage), Irelia (guerrilla fighter, good sustain, can disengage from fights).

5. Game Data

Combat:

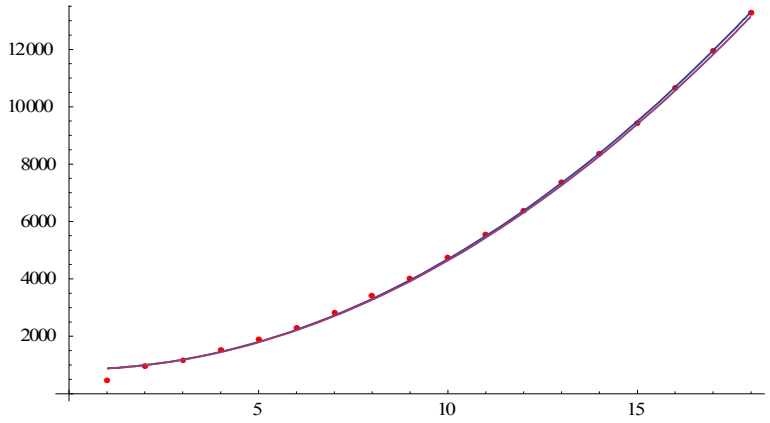
http://leagueoflegends.wikia.com/wiki/Armor_penetration

http://leagueoflegends.wikia.com/wiki/Armor

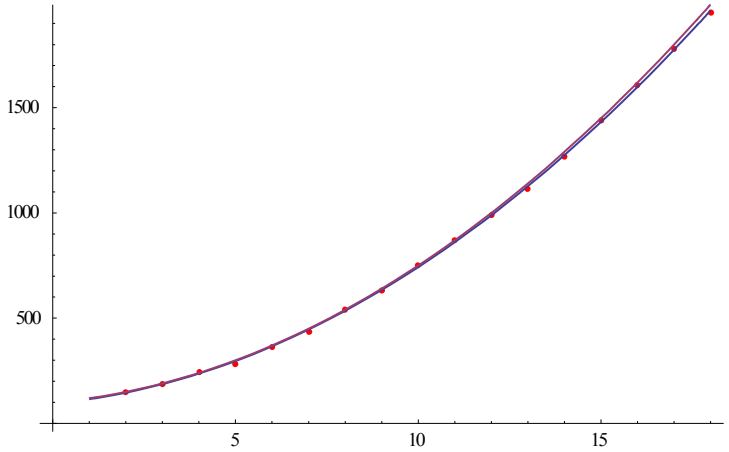
http://leagueoflegends.wikia.com/wiki/Attack_speed

```
armorMitigation = 100/(100+(xArmor*xArmorPenP))

autoAttackDPS = armorMitigation * xAttack * xCriticalStrike * xAttackSpeed

xCriticalStrike = 1 + (1 * Min[1,bonusCritChance])]

xAttackSpeed = Min[2.5, (0.625/(1+character[AttackSpeedDelay])) * (1 +
bonusAttackSpeed)
```

Rough approximation of gold earned and time elapsed (seconds):




```
goldAtLevel[lv_] := 840 + 38lv^2
```




```
timeAtLevel[lv_] := 100 + 15lv + 5lv^2
```

Cost per stat:

http://leagueoflegends.wikia.com/wiki/Gold_efficiency

```
costAttack = 36
costHealth = 2.64
costArmor = 20
costAbilityPower = 21.75
costCriticalChance = 50*100
costAttackSpeed = 33.33*100
```

Random character's stat scaling:

http://leagueoflegends.wikia.com/wiki/Irelia

```
irelia[HealthBase]       = 456;
irelia[HealthLv]         = 90;
irelia[AttackBase]       = 56;
irelia[AttackLv]         = 3.3;
irelia[ArmorBase]        = 15;
irelia[ArmorLv]          = 3.75;
irelia[AttackSpeedBase]  = 0.625;
irelia[AttackSpeedDelay] = -0.06;
irelia[AttackSpeedLv]    = 0.0322;
```