32

Paragon Bots

A Bag of Tricks

Mieszko Zieliński

32.1 Introduction

Paragon is a MOBA-type game developed by Epic Games, built with Unreal Engine 4 (UE4). Relatively late in the project a decision was made to add AI-controlled players (a.k.a. bots) into the game. Limited time and human resources, and the fact that crucial game systems had already been built around human players, meant there was no time to waste. Redoing human-player-centric elements was out of the question, so the only way left to go was to cut corners and use every applicable trick we could come up with. This chapter will describe some of the extensions we made to the vanilla UE4 AI systems as well as some of the simple systems tailored specifically for *Paragon* player bots. In the end, we added a few enhancements to our basic behavior tree implementation, came up with a few useful MOBA-specific spatial representations, integrated all of those with our mature spatial decision-making system, the environment query system (EQS), and added a few other tricks. The results exceeded our expectations!

32.2 Terms Primer

There are a number of terms in this chapter that might be unknown to the reader, so we will explain them here.

In *Paragon*, players control heroes. A *hero* is a character on a team that can use *abilities* to debuff or deal damage to enemy characters and structures, or to buff his or hers own teammates. Buffing means improving capabilities (like regeneration, movement or attack speed, and so on) whereas debuffing has an opposite effect. Abilities use up hero's *energy*, which is a limited resource. Abilities also use up time, in a sense, since they are gated by cooldowns. Some abilities are always active, whereas others require explicit activation. The goal of the game is to destroy the enemy's base, while protecting your own team's base. Defending access to the base are *towers* which teams can use to stop or slow advancing enemies. The towers are chained into paths called *lanes*. A lane is what *minions* use to move from one team's base to the other's. Minions are simple creatures that fight for their team, and they are spawned in *waves* at constant intervals.

There is more to the game than that; there is jungle between lanes, inhabited by jungle creeps (neutral creatures that heroes kill for experience and temporary buffs), where *experience wells* can be found. There is the hero and ability leveling up, and cards that provide both passive and active abilities, and much more. However, this chapter will focus only on how bots wrapped their heads around game elements described in the previous paragraph.

32.3 Extremely Parameterized Behavior Trees

An experienced AI developer might be surprised that all *Paragon* bots use the same behavior tree. With the time and resources constraints we were under, we could not afford to develop separate trees for every hero, or even every hero type. This resulted in a specific approach: The master behavior tree defines the generic structure, specifying the high-level order of behaviors, but details of behavior execution (like which ability to use, which spatial query to perform, and so on) are parameterized so that runtime values are polled from the AI agent when needed.

32.3.1 Vanilla UE4 Behavior Trees

Before we get into details of how Vanilla UE4 Behavior Trees (BTs) were used and expanded in *Paragon*, here is a quick overview. Since BTs have been in AI programmers' toolkit for years (Isla 2005) the description will be limited to what the UE4 implementation adds to the concept.

UE4 BTs are an event-driven approach to generic BTs. Once a leaf node representing a task is picked the tree will not reevaluate until the task is finished or conditions change. Execution conditions are implemented in the form of *decorator* nodes (Champandard 2007). When its condition changes, a decorator node may abort lower priority behaviors or its own subtree, depending on the setup.

The UE4 BT representation is closely tied to UE4's Blackboard (BB). In UE4, the blackboard is an AI's default generic information storage. It takes the form of a simple key-value pair store. It is flexible (it can store practically any type of information) and has a lot of convenient built-in features. Blackboards are dynamic in nature and are populated by data

at runtime. BB entries are the easiest way to parameterize behavior tree nodes; it makes it possible for a BT node requiring some parameters to read the values from a blackboard entry indicated by a named key. BT decorator nodes can register with BB to observe specific entries and react to stored value changes in an event-driven fashion. BB entries are also used to parametrize BT nodes. One example of parametrized BT nodes is the *MoveTo* node, which gets the move goal location from a blackboard entry.

Our BT implementation has one more auxiliary node type—the *service* node. It is a type of node that is attached to a regular node (composite or leaf) and is "active" as long as its parent node is part of the active tree branch. A service node gets notification on being activated and deactivated, and has an option to tick at an arbitrary rate.

32.3.2 Environment Querying System

In UE4, the EQS is the AI's spatial querying solution and is mentioned here since it is mostly used by the BTs to generate and use runtime spatial information. EQS is used for tasks such as AI positioning and target selection (Zielinski 2013). EQS' queries are built in the UE4 editor, using a dedicated tool, and are stored as reusable templates. The vanilla UE4 BT supplies a task node and a service node for running EQS queries and storing the results in the blackboard.

For *Paragon*, we made changes to the EQS so that it would be possible to point at a query template we want to use by specifying a key in the blackboard. The query templates themselves are regular UE4 UObjects, so no work on the blackboard side was required. The only thing that needed to be done was to extend the BT task that issues environmental queries to be able to use query templates indicated by blackboard values. We then used this new feature to implement different positioning queries for melee and ranged heroes; melee heroes want to be very close to enemies when attacking, whereas ranged ones (usually) want to be at their abilities' range while keeping their distance so that the enemy does not get too close.

32.3.3 Blackboard Extension

Allowing the blackboard to store a new type is as simple as implementing a dedicated BB key type. For *Paragon* we added a dedicated key type for storing an *ability handle*, a value that uniquely identifies an ability the given hero could perform. With the new blackboard key type, we gained an easy way to configure BT nodes to use abilities picked for different purposes. Section 32.4 describes the way abilities are picked.

32.3.4 Behavior Moods

It is easy to think about the behavior tree as the final consumer of AI knowledge. The BT takes the data and decides on the best behavior, based on that knowledge. We do, however, have additional subsystems that need information regarding what is going on in the BT. It is not really about what the BT is doing specifically, just what its current "mood" is. We need to know if the bot is running away, attacking characters, attacking towers, and so on.

The current mood is set through a dedicated service node. The mood information is then used by some of the native code that is doing derived work, like setting AI agent's focus or deciding which movement-related abilities are allowed.

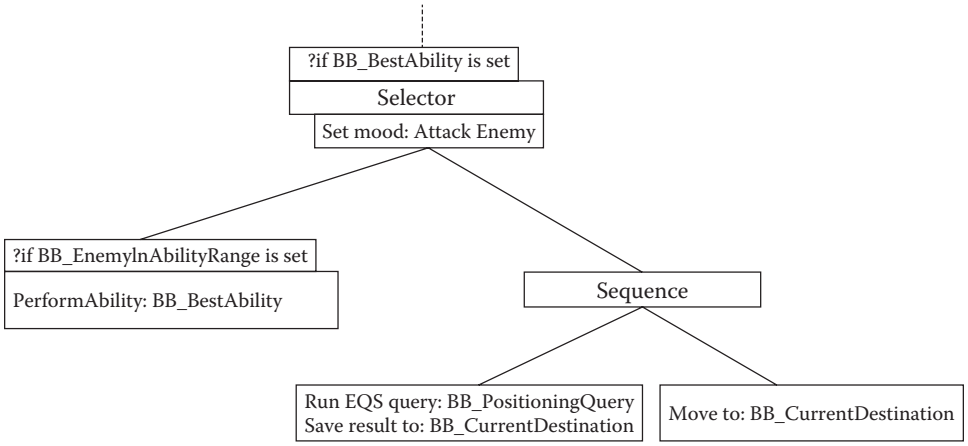

Figure 32.1

Example behavior tree branch controlling use of offensive abilities.

An example of how the described extensions and features mix together is shown in Figure 32.1. Names prefixed with BB indicate blackboard entries.

32.4 Ability Picker and Bot Ability Usage Markup

Deciding which ability to use in a given context, against a given target, is not a trivial task. The decision depends on multiple factors, like the target type, what we want to do to it, how much energy we have available at the moment, and which abilities are on cooldown. In addition, many abilities have multiple different effects. For example, there are abilities that damage or slow enemies, but they may also heal friendly heroes.

Abilities in *Paragon* are defined in UE4's Blueprint visual scripting language. This gives designers great flexibility in terms of how exactly each ability executes its behavior. Although this is great for content creator iteration and expressing creativity, it makes it practically impossible to extract the ability's usage information automatically. Besides, raw information about an ability is not enough to figure out how and when it makes sense to use it. To address that, we came up with a family of tags that designers used to mark up every ability an AI is supposed to use. We called those *bot ability usage* tags; examples are shown in Table 32.1.

Table 32.1 Examples of Bot Ability Usage Tags

BotAbilityUsage.Target.Hero	Use given ability to target heroes.
BotAbilityUsage.Effect.Damage	Given ability will damage the target.
BotAbilityUsage.Effect.Buff.Shield	Given ability will give the target a shield.
BotAbilityUsage.Mobility.Evade	Given ability can be used to evade enemy attack.

32.4.1 Ability Cached Data

When a bot-controlled hero is granted an ability, we digest the ability's blueprint to extract information useful for the AI. We store the results in the bot's *Ability Cached Data*, putting a lot of care into making sure the data representation is efficient. Ability usage tags get digested and represented internally as a set of flags. Other information stored includes the ability's range, type, damage, energy use, cooldown length, and so on. It also caches timestamps indicating when the given ability's cooldown will be over and when AI will have enough energy to cast the given ability. Every active ability available to a bot-controlled hero has its representation in the ability cache. Ability cached data is the key to the Ability Picker's efficiency.

32.4.2 Ability Picker

The *Ability Picker* is a simple yet surprisingly powerful service that is responsible for picking the right ability from a set of given abilities, given a certain target. The way the Ability Picker does that is extremely simple—actually, all the magic is already contained in the way the data is digested and stored as *ability cached data*. All the Ability Picker does is iterate through the list of abilities usable at a given point in time and checks if it matches the desired effect and target type. The returned ability is the one of "best cost" among the ones applicable. "Best cost" can have different meanings depending on the target type. When targeting minions, we prefer cheaper abilities, whereas we save the more expensive ones to target the heroes. Needless to say this scoring approach leaves a lot of room for improvement.

The core Ability Picker algorithm is extremely straightforward and is presented in pseudocode in Listing 32.1.

Listing 32.1. Ability Picker's core algorithm.

```
FindAbilityForTarget(AIAgent, InTargetData, InDesiredEffects)
{
    BestAbility = null;

    for Ability in AIAgent.AllAbilities:
        if Ability.IsValidTarget(InTargetData)
            && (Ability.DesiredEffects & InDesiredEffects)
            && (Ability.CooldownEndTimestamp < CurrentTime)
            && (Ability.EnoughEnergyTimestamp < CurrentTime):
                Score = Ability.RequiredEnergy;
                if IsBetterScore(InTargetData, Score, BestScore):
                    BestScore = Score;
                    BestAbility = Ability;

    return BestAbility;
}
```

`AllAbilities` is an array containing ability cached data of every ability available to the bot. The `IfValidTarget` function checks if a given target is of an appropriate type (Hero, Minion, Tower), if it is of a valid team, and if the target's *spatial density* (described below) is high enough. `IsBetterScore`, as mentioned above, prefers lower scores for minions and higher scores for heroes, so that we go cheap while fighting minions and wait to unload on heroes.

32.4.3 Target's Spatial Density

Some abilities are tagged by designers as usable against minions, but it makes sense to use them only if there is more than one minion in the area. This applies to *Area-of-Effect* abilities (AoE), which affect multiple targets in a specified area rather than a single target. Using such an ability on a single minion is simply a waste of energy and time.

To be able to efficiently test if a given target is "alone," we find *Spatial Density* for every target we pass to the Ability Picker. Target density is calculated as part of influence map calculations, which is described later in this chapter, so getting this information at runtime is a matter of a simple lookup operation.

32.4.4 Debugging

Having one system to control all ability selection had an added benefit of being easier to debug. It was very easy to add optional, verbose logging that once enabled would describe which abilities were discarded during the selection process, and why. The logged information combined with the spatial and temporal context we get out of the box with UE4's Visual Log allowed us to quickly solve many ability selection problems—which usually turned out to be bugs in ability markup. You can never trust those darn humans!

A handy trick that proved invaluable during bots' ability execution testing was adding a console command used at game runtime to override ability selection to always pick the specified ability. Thanks to the centralized approach we were able to implement it by plugging a piece of debugging logic into Ability Picker's `FindAbilityForTarget` function that would always pick the specified ability.

32.5 One-Step Influence Map

The influence map is a concept well known in game AI; it has been around for many years (Tozour 2001). It is a very simple concept, easy to grasp, straightforward to set up, but produces great, useful data from the very simple information it is being fed. The idea is based on a notion that units exert a "spatial influence" on their environment, proportional to their strength, health, combat readiness, or anything else that decays with distance. The influence map is a superposition of all those influences and can be used to guide AI decisions.

Normally, building an influence map involves every agent going through two steps. First is to apply the agent's influence at the agent's current location. This usually is the place where the agent has the highest influence (although there are other possibilities [Dill 2015]). The second step is influence propagation. We take the given agent's influence and propagate it to all neighboring areas, and then to areas neighboring those areas, and so on. The agent's influence distributed this way is a function of distance—the further from the source, the weaker the influence is.

Influence propagation can be a very expensive operation; depending on the influence map representation and resolution (although there are algorithms supplying infinite-resolution influence maps [Lewis 2015]). Also, it gets even more expensive the more influence sources we consider. Due to the constraints on processing for *Paragon* servers, the naive approach was not chosen.

There are multiple ways to represent an influence map. Performance is very important to *Paragon* bots, so we went for a very simple structure to represent influence on our maps. Since there is no gameplay-relevant navigable space overlaps on the vertical axis, we were able to represent the map with a simple 2D cell grid, with every cell representing a fixed-size square of the map. The size of the square used was a compromise between getting high-resolution data and not taking too much memory to store the map or using too much CPU when calculating influence updates. After some experimentation, we settled on using cells of 5×5 m which was a good compromise between memory requirements (320 kB for the whole map) and tactical movement precision. In addition, we have auxiliary information associated with every cell where we store influence source's counters that are used in multiple ways. More on that later.

We cannot simply ignore the fact that different enemies have different range and strength properties, that would affect the influence map naturally with influence propagation. In lieu of actual influence propagation, we apply influence of some agents to map cells in a certain radius rather than just in the one cell where the agent is currently present. We used zero radius for every minion (even the ranged ones) and for heroes we used every given hero's primary ability range. One could argue that applying influence in a radius rather than a point is almost the same as influence propagation, but there is a substantial performance gain when applying the influence to every cell in a radius compared to propagating it to consecutive neighbors, especially if the propagation would care about cell-to-cell connectivity. Applying influence to all cells in a radius does have a side effect of ignoring obstacles that would normally block influence, but due to the dynamics of *Paragon* matches and the way *Paragon* maps are built, this effect is negligible.

The main way we wanted to use the influence map was to determine bot positioning in combat. Depending on the hero type and situation, we might want a bot to keep away from enemies (the default case for ranged heroes), or on the contrary, keep close to enemies (the default for melee heroes). We can also use "friendly influence" as an indication of safer locations, or the opposite, to help bots spread out to avoid being easy AoE attack targets. It turns out that influence propagation is not really required for the described use cases because the influence range, defined as heroes' effective range, is already embedded into influence map data. Propagated data would give us some knowledge regarding how the tactical situation may change, but in *Paragon* it changes all the time, so we went for a cheaper solution over the one that would produce only subtly better results. Influence propagation can also be faked to a degree by deliberately extending the radius used for every hero. The extension can even be derived from runtime information, like current speed, amount of health, energy, and so on, because we build the influence map from scratch on a regular basis.

As will be discussed below, the influence map integrates with EQS to impact spatial processes like positioning, target selection, and so on.

32.5.1 Other Influence Sources

Other game actor types can also alter the influence map. Let us first consider towers (defensive structures described in Section 32.2). All characters entering an enemy tower's

attack range are in serious danger, since towers pack a serious punch, even to high-level heroes. However, influence information is being used only by hero bots, and heroes are safe inside enemy tower range as long as the hero is accompanied by minions—minions are the primary target for towers. For this reason, we include a given tower's influence information in the map building only if the tower has no minions to attack; otherwise the bot does not care about the tower danger (or in fact, even know about it!).

One thing worth mentioning here is that since towers are static structures we do not need to recalculate which influence map cells will be affected every frame. Instead, we gather all the influenced cells at the start of the match and then just reuse that cached information whenever we rebuild the influence map.

One other influence source we consider during influence map building is AoE attacks. Some of those attacks persist long enough for it to make sense to include them in influence calculations. Having that information in the influence map makes it easy to "see" the danger of going into such an area! We do not annotate the influence map with short-lasting AoE attacks since the AI would not have a chance to react to them anyway—those attacks last just long enough to deal damage and there is practically no chance to avoid them once they are cast.

32.5.2 Information Use

As stated previously, the main use of the influence information is for bot positioning. This information is easily included in the rest of positioning logic by adding another test type expanding our spatial querying system (EQS). Thanks to EQS test mechanics, a single test that is simply reading influence information from specified locations in the world can be used to both score and filter locations a bot would consider as potential movement goals. Incorporating this one simple test into all bots' positioning queries allowed us to get really good results quickly. Thanks to this change, bots gained the power to avoid entering enemy towers' fire or running into groups of enemies and to pick locations close to friends, and so on.

Recall the auxiliary information associated with every cell of the map. That information is not strictly part of the influence map, but it is gathered as part of influence map building. The auxiliary information includes a list of agents influencing each cell. We use this information to improve the performance of minions' perception by effectively reducing the number of targets they consider for regular line-of-sight tests. Querying the influence map for enemy minions or heroes in a given area boils down to a simple lookup operation.

One last bit of influence-derived data is something we called *target density*. It is a simple per-cell counter of enemies of a given type (minion or hero), and we use that to determine if a given target is "alone" or if we would potentially hit some other targets when attacking the specific considered target. This is the information that hints to the Ability Picker whether using an AoE ability on a given target would be a waste of *energy* or not.

This kind of creative data reuse was necessary due to our time restrictions. We spent time building a system, so then we had to squeeze as much from it as possible.

32.6 Lane Space

A question we often had to answer was "how far bot *X* is from *Y* in terms of the lane it is on," where *Y* could be an enemy tower, a hero, a minion wave, or just an arbitrary location in the world. We did not really care about actual 3D distance, just about "how far along

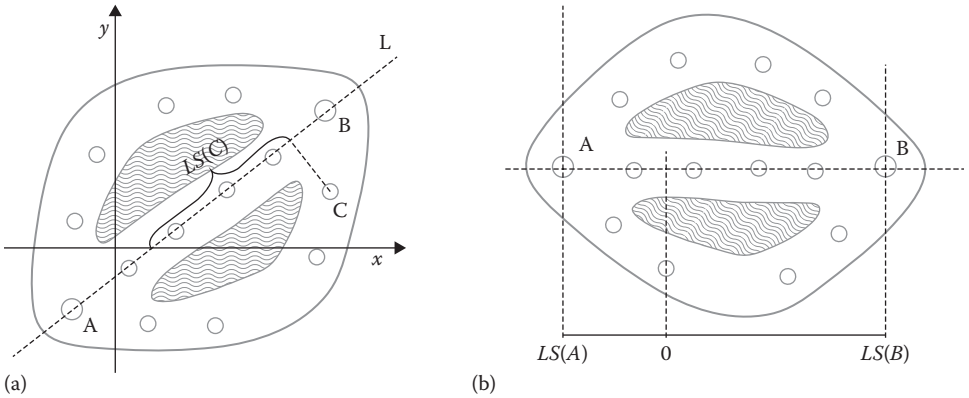

Figure 32.2

(a) A traditional MOBA map. (b) Map transformed to align with one of the axes.

the lane" things were. This is extremely easy to answer if lanes are straight lines, but that is not quite the case in *Paragon*… unless we straighten them out!

Figure 32.2a depicts a regular MOBA map, where A and B mark locations of both teams' bases. There are three lanes, with left and right lanes being not at all straight. This is where the concept of *Lane Space* comes in. Transforming a 3D world location into the lane space is a matter of projecting it to the line defined by $\overline{AB}$ segment. See Figure 32.2b for an illustration of this transformation.

Introducing lane space made it easy to add a new type of EQS test for scoring or filtering based on relative distance to the combat line on a given lane. We used it for example to make ranged bots prefer locations 10 meters behind the minions.

32.6.1 Lane Progress

A natural extension of the lane space is the idea of *Lane Progress*. It is a metric defined as a location's lane space distance from A or B, normalized by $|\overline{AB}|$ (distance from A to B). Lane progress is calculated in relation to a given team's base, so for example for team A lane progress value of base A location would be 0, and base B location would be 1. For team B it is the other way around; in fact, for every team A's lane progress value of x, the value for team B will be equal to $(1 - x)$.

32.6.2 Front-Line Manager

It is important in a MOBA for heroes to understand positioning in a lane based on areas of danger and safety. In combat, the ranged heroes prefer staying a bit behind the minion line, whereas melee heroes should be at the front-line, where the brawling takes place.

To allow the bots to behave like real humans do, we created the Front-Line Manager. Its sole purpose is to track all the minions left alive, along with all the remaining towers, and calculate where the lane's combat is. The Front-Line Manager is being fed information regarding minions by the influence map manager during influence map building. Based on that information and on the current state of a given team's towers on every lane, the Front-Line Manager is calculating the *front-line* on every lane for both teams. The exact front-line value is expressed in terms of lane progress.

Similarly, to the influence map information, we incorporate front-line information into the positioning logic by implementing another EQS test that measures locations' distance to the front-line.

One other place the front-line information is relevant is during enemy selection. We want to avoid bots chasing enemy heroes too deep into the enemy territory, so the distance from the front-line contributes to target scoring. Again, this is done with a dedicated EQS test.

32.7 Other Tricks

Cutting corners was crucial due to time constraints, so there is a fair amount of other, smaller tricks we used. Those are usually simple temporary solutions that either work well enough so that players would not notice, or are placeholders for solutions that will come in the future.

The first one worth mentioning is *perfect aim*. The ranged hero bots aim exactly at their target's location. This is not strictly speaking cheating, since experienced players do not have a problem doing the same. And it is not even as deadly as it sounds, since most ranged abilities used physical projectiles (meaning the hits are not instant) and some of them have ballistic properties (meaning they obey gravity). It is not a problem to add a slight aim angle deviation; *Paragon*'s easy-difficulty bots actually do that, it is just that the "perfect aim" helps revision-one bots to bridge the skill gap to human players. Besides, there are so many things humans have brain-hardware support for (especially visual processing), why should bots give up one of the few things bots are born with!

Another simple trick we used was to address a complaint we got from people playing in mixed human-bot teams. The problem was that as soon as the game started, all the bots were taking off to race down the lanes. It was suggested that bots should wait a bit, for example until minion waves started spawning. Since that would be a one-time behavior, considering it as a part of regular AI reasoning would be a waste of performance. Good old *scripted behavior* came to the rescue. We came up with a very simple idea (and implementation) of a one-time behavior that is triggered at the beginning of the match. It makes bots wait for minions to spawn and then *flow* down the lanes until they've seen an enemy, or reached the middle of the map, at which point bots simply switch over to the default behavior. *Flowing* down the lane involves using *Paragon*'s custom navigation flow-field, which makes movement fully pathfinding-free, and thus a lot cheaper than regular AI navigation. Once we had scripted behavior support, it came in useful in testing as well.

32.8 Conclusion

I feel that working on game AI is an art of using what is available and coming up with simple solutions to usually not-so-simple problems. In the chapter we've shown how this approach has been applied to work done on *Paragon* bots. Reusing and extending your AI systems is especially crucial when working under heavy time pressure, so investing effort ahead of time to make those systems flexible will pay off in the future. Using a single behavior tree for all bots in *Paragon* would not be possible otherwise. When it comes to solving game-specific problems, it is usually best to come up with a simple solution that isolates the problem and hides the complexity from the rest of AI code by supplying some easy-to-comprehend abstraction. The Ability Picker and Front-Line Manager are great examples of this. The "Keep It Simple" rule is always worth following!

References

Champandard, A., Behavior trees for Next-Gen AI, *Game Developers Conference Europe*, Cologne, Germany, 2007.

Dill, K., Spatial reasoning for strategic decision making. In *Game AI Pro 2: Collected Wisdom of AI Professionals*, ed. S. Rabin. Boca Raton, FL: A. K. Peters/CRC Press, 2015.

Isla, D., Handling complexity in the Halo 2 AI, *Game Developers Conference*, San Francisco, CA, 2005.

Lewis, M., Escaping the grid: Infinite-resolution influence mapping. In *Game AI Pro 2: Collected Wisdom of AI Professionals*, ed. S. Rabin. Boca Raton, FL: A. K. Peters/CRC Press, 2015.

Tozour, P., Influence mapping. In *Game Programming Gems 2*, ed. M. Deloura. Hingham, MA: Charles River Media, 2001.

Zielinski, M., Asking the environment smart questions. In *Game AI Pro: Collected Wisdom of AI Professionals*, ed. S. Rabin. Boca Raton, FL: A. K. Peters/CRC Press, 2013.