Markov Decision Processes

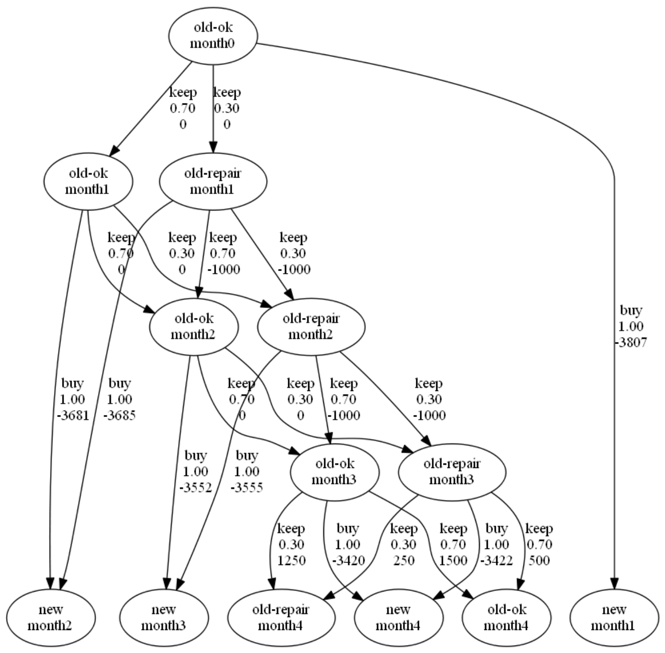
Jonathan Hudgins

Buying a Car

I purchased a 2001 Volvo station wagon more than 5½ years ago. It has been a constant source of turmoil because, although I have paid it off, it frequently needs repair. Should I trade it in for a new car, or continue to repair it? This can be modeled as a simple Markov Decision Process where the decision to repair the old car or purchase a new car can be made each month.

This problem is particularly relevant (I'm thinking of buying a new car now) and simple for implementing and debugging our solution algorithms.

States	For each month: ***old-ok*** (doesn't need repair)*,* ***old-repair*** (needs repair)*,* ***new***
Actions	***keep****,* ***buy***
Transitions	***keep*** actions from ***old-ok*** or ***old-repair*** can both transition to ***old-ok*** or ***old-repair*** ***buy*** always transitions to ***new***
Rewards	No reward for ***keep*** – from ***old-ok*** Negative expected repair cost for ***keep*** from ***old-repair*** Negative cost for remaining months for ***buy***
Terminal	***buy*** for any month, ***keep*** after specified month with salvage reward



Buy Car MDP

The graph on the left shows states and transitions with corresponding actions, probabilities, and rewards if we limit the model to 5 months.

Note: We display rewards as transition-specific (which include state rewards for simplicity).

There is no need to model states after car purchase, because there are no more decisions to be made (at least, we hope, not for years to come). We simply use the calculated cost of owning the car for the remaining months being simulated.

In particular, the cost of owning the new car is the sum of: expected depreciation, interest, new car tax, new car registration, and expected repair cost. Because the car is an asset, the purchase price or monthly payments don't directly matter if we capture initial overhead (tax, registration), depreciation and interest. Each monthly cost, however, must also be adjusted by our discount reward rate used in our learning algorithms. This adjustment will provide a fair comparison between the early terminated cost and the final months.

There are three states for every month (except for the first). So there will be a total of *3*(*m-1) + 1* states – where *m* is the month. Each ***old*** car state (except for the terminal) has 2 actions and 3 transitions for a total of *4m-6* actions actions and *6m-9* transitions. A 12 month model (used as default) has 34 states, 42 actions and 63 transitions.

To calculate expected repair costs, we look at the recent history. The last 33 months have had 10 repairs for a total of $10,125. So we use an expected repair cost of $1000 with a repair-needed probability of 0.3. For the cost of buying we use a purchase price of $24698, an interest rate of 3.1%, low depreciation (immediate: 5%, annual:15%), and high trade in value of $3000 (the car is old). We reduce trade in value by half of repair cost if repair is needed (dealerships place almost no value on repairs). For the terminal states still owning the old car, we use half of the trade in value as salvage.

Basketball End Game

The final 60 seconds of a basketball game can be one of the most exciting sports to watch or play in. From buzzer-beaters to heart-breakers the basketball end-game is full of drama. Much of the drama comes from the interplay between team strategies, the probability factor, and the extra value of a three point shot. It is a 2 player zero-sum game – and while there are points at every state, the only true reward is whether a team wins or loses.

When should a team shoot or wait? Should a team go for the 3 point shot or 2 point?

A solution to this dilemma would be extremely valuable to coaches and players and could make teams more successful.

The below model makes these simplifying assumptions:

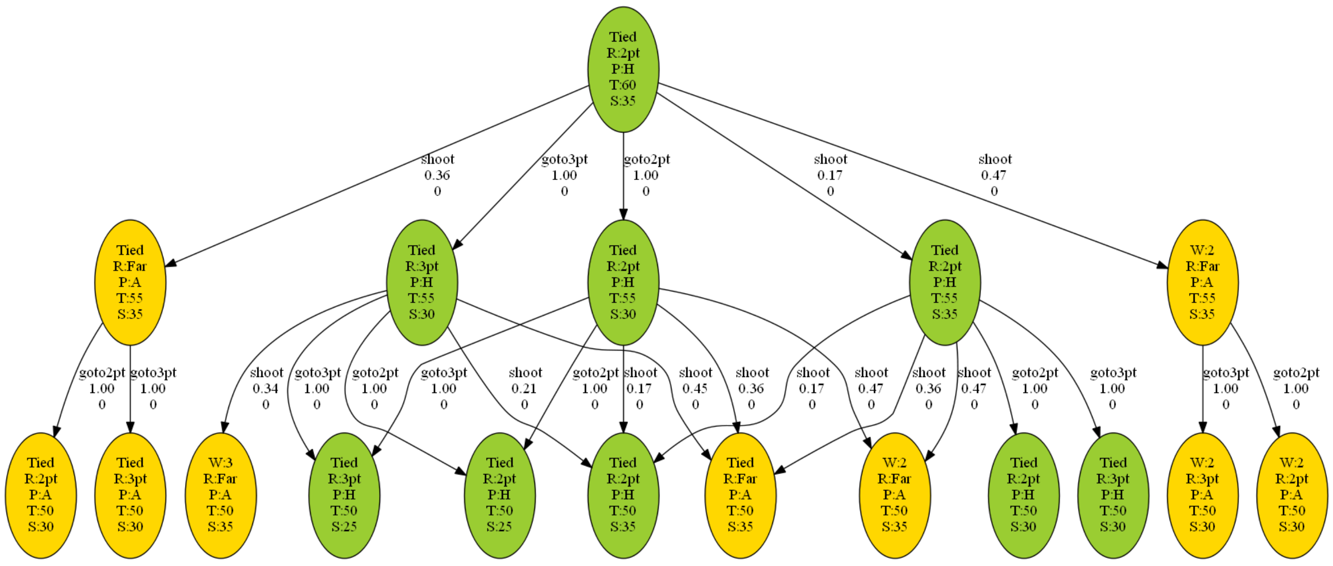
- Each action occurs in a 5 second interval
- Fouling and foul shots are not considered
- Good/bad shots are not modeled (assume that in a 5 second interval a reasonable shot can be found)
- Shot percentages are the same for all individuals on each team (and same at end of game as overall averages)
- Steals are not modeled
- Offensive and defensive rebounding is modeled based on season averages
- Absolute points are irrelevant, only point differential matters
- Shot must be made at last increment before final time (that is, 5 seconds left on shot or game clock)
- More than *n* point differential (default to 5) is considered a win

States	For each 5 second increment: • ***point-differential***: [-5, 5] • ***shot-range***: [too-far, 3pt, 2pt] • ***possession***: [home, away] • ***time-remaining*** • ***shot-clock***
Actions	***shoot***, ***goto-3pt, goto-2pt***
Transitions	• ***shot-clock*** and ***time-remaining*** decremented by 5 seconds • successful ***shoot*** results in ***point-differential*** up/down by 2/3 for ***home***/***away***, **2pt**/**3pt** and change of ***possession*** • failed ***shoot*** results in either: • offensive rebound, reset ***shot-clock*** and no change of ***possession*** • defensive rebound: reset ***shot-clock*** and change of ***possession***
Rewards	1000 for win, 0 for tie, -1000 for loss
Terminal	If points are over/under max/min ***point-differential,*** team with more points declared winner When ***time-remaining*** is 0, team with more points declared winner

Probability of a successful shot is determined by 2 point and 3 point shooting percentages (NCAA overall: 47.0% and 34.4% respectively). Probability of offensive/defensive rebound is the probability of missing times percentage of offensive/defensive rebounds (NCAA overall: 31.5% offensive, 68.5% defensive). The shot clock resets to 35 seconds (NCAA rules)

Because this is a 2-player zero-sum game, the algorithms will maximize/minimize discounted utility based on ***possession***. Each 5 second increments will have *11 * 3 * 2 * 7 = 462* possible states. For the 12 increments in

60 seconds, the total possible number of states is 5544. In order to simplify the problem we analyze a single initial state (home ***possession, tied*** with 60 seconds ***time-remaining***, 2pt ***shot-range*** and full 35 second ***shot-clock***). This initial state results in an MDP with 674 states, 1872 actions, 2896 transitions. The first three levels of this MDP are shown below:



Basketball MDP First 3 Levels

Starting on the far right, the above graph shows our first choice of ***shoot*** with a 0.47 probability of scoring – which changes ***possession*** (home is green, away is yellow). ***shoot*** can also result in a miss with a defensive rebound – far left (miss has 0.53 probability, defensive rebound gets 68.5% of rebounds for total probability of 0.36). ***shoot*** can result in a third transition of a miss with an offensive rebound (0.17 probability). Other actions are ***goto-3pt*** or ***goto-2pt,*** which change state with 1.0 probability. After the next level, the graph gets too busy to represent in this paper.

Value and Policy Iteration

Implementation

For Value Iteration the algorithm (implemented in Python) simply iterates through all states and their actions, updating expected utility for all actions and possible transitions with discounted reward (gamma = 0.99). For BuyCar updated utility is simply the maximum for all actions. For Basketball the algorithm chooses maximum/minimum based on home/away ***possession***. Iteration terminates after utility converges (within tolerance of 0.001) or 100 iterations.

For Policy Iteration the algorithm starts with a random policy. Then the algorithm iterates on evaluating utility of that policy and find the new policy based on that utility. To evaluate the policy, a NxN matrix, *T,* is created (where N is the number of states) for the probability of transitioning from state *i* to state *j* (states are simply arbitrarily indexed for simplicity). A Nx1 matrix, *R,* is created for the total expected reward at state *i.*
To find *V* (vector of utilities for each state *i*) solve the equation:

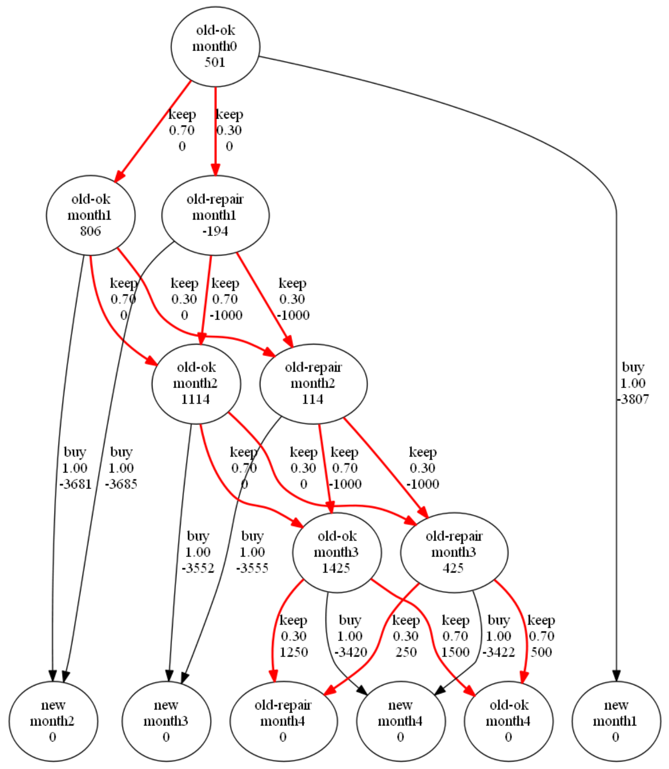
$$V = \gamma T \times V + R \qquad \text{solve for V} \qquad V = [I - \gamma T]^{-1} \times R$$

We use Python's numpy module to create the matrices, calculate the inverse and solve for V. While this is very convenient, it is very slow. It is particularly slower than need be, because most of the entries are 0 (each row has at most 4 entries for each possible transition). The best general matrix inverse algorithms calculate in $O(n^{2.37})$ and same for matrix multiplication. For simplicity, solving linear systems is often referred to as having $O(n^3)$ complexity. Either way when n is even just 674 states, solving the linear system can require significant time.

Comparisons

Both Value Iteration and Policy Iteration converge on the same utility values for each state (give or take a small precision difference). This can be easily verified by printing out the utilities and diffing the files (see output/basketball.*.utilities and output/buycar.*.utilities).

Buying Car Analysis



Buy Car 5 Month Value/Policy Iteration

The Buy Car graph on the left (run for 5 months) is identical for Value and Policy Iteration. The optimal policy (calculated by VI and PI) is shown in red. And it quite simply chooses to keep the old car. The cost of frequent repair simply cannot match the cost of depreciation. This pattern continues for the full 12 months – and even if the analysis is run for 24 months.

This seems like an uninteresting straight-forward decision. But changing some of the numbers can be revealing.

When the average repair cost is increased to $2000 the policy recommends ***buy*** if the car needs repairs in the first 4 months of a 12 month simulation (see output/buyCarRepair2000.png if interested). After that the policy recommends always keeping the car, whether repair is needed or not.

Changing the length of the simulation also changes the recommendation (see output/buyCar36Months.png if interested). A 36 month simulation will recommend ***buy*** if repairs are needed in the first 12 months. Afterward it is more advantageous to always ***keep,*** repairing the car if needed.

Changing the salvage reward also influences the decision making process. With zero salvage cost and 18 months, the policy recommends ***buy*** in the first month if repairs are needed. All other months recommend ***keep*** regardless of repair state.

Another interesting phenomenon is that as we move closer to the initial state the utility of an ***old-ok*** and ***old-repair*** is closer. In a 36 month simulation the difference is 1000 at month 12 or more, and 437 at month 1. When the policy recommends ***buy***, the utility of ***old-repair*** state is determined by the ***new*** state utility whereas ***old-ok*** is still determined by repair costs which produce worse utility the further back in time we run the simulation.

It seems that maybe the salvage reward influences the values closer to the end state. Or, with shorter time, the initial registration, tax cost and initial depreciation dominate. Both result in the ***keep*** policy – even if repair is needed. But if the repair cost is greater, the salvage is reduced or the simulation is carried out for a long time, the optimal policy becomes ***buy*** (when repair is needed). Repairing the car is good for the short term but a long term loss – which is counter-intuitive to the old-fashioned mentality of repairing things until they are all used up!

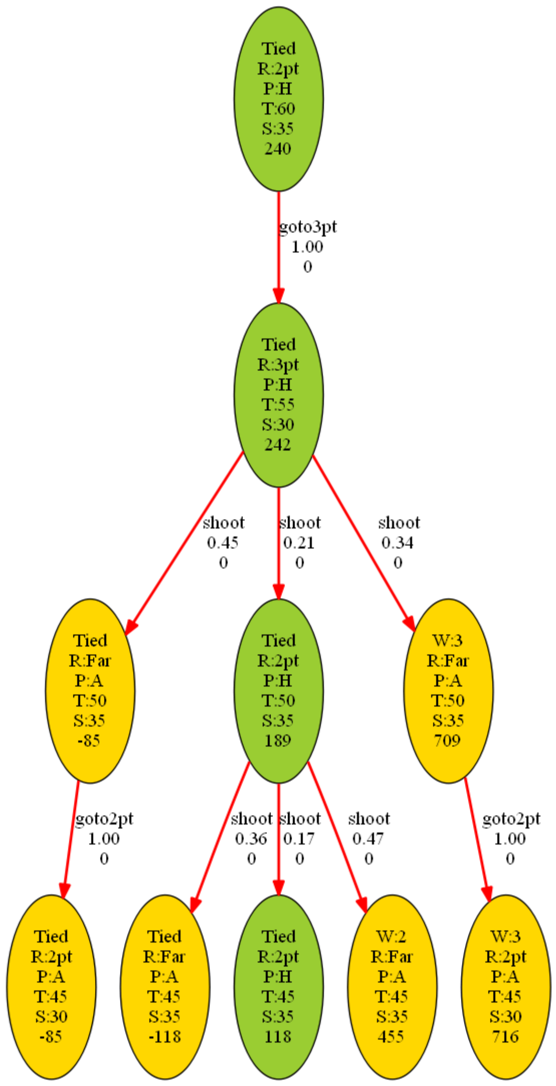
Basketball Analysis

The graph on the following page shows the first 4 levels of the optimal policy. Non-policy nodes are pruned from the graph for display simplicity.

The first recommendation is to ***goto-3pt*** and then ***shoot***. However, on an offensive rebound (middle branch), the optimal policy recommends ***shoot*** at the ***2pt-range***, instead of going for a 3pt shot! This means deciding between the 3pt vs. the 2pt shot is time sensitive. With 60 or 55 seconds left, the 3pt shot is better. But with 50 seconds left (and still tied), it is better to take the 2pt shot (instead of setting up a 3pt shot or waiting).

Once the game reaches 45 seconds remaining (level 4), there is another twist. For all states (not shown in the graph below) the optimal policy is to wait (***goto-3pt*** or ***goto-2pt***) and doesn't change until 30 seconds remaining. This waiting period is recommended regardless of point differential or possession.

Note: the following states aren't shown in the graph (but the graph output/basketball.optimalonly.png demonstrates the detail).



Basketball Optimal Policy 4 Levels

At 30 seconds, the optimal policy recommends the away team to shoot a 3 pointer if they are down by 3 points. However, if they are only down by 2 points, the optimal policy recommends waiting until 15 seconds left and then shoot the 3 point shot. I'm not entirely sure why these are different. But a major difference is that the chances of winning when down by 3 points are slim (tie more likely), but the team still has a reasonable chance of winning when down by 2 points.

There is another interesting situation. A common strategy is to shoot with enough time so that even if your opponent uses the entire shot clock, you will get the ball back with enough time to attempt another shot. But an easily overlooked consideration is the possibility of the rebound. Any shot taken with 10 or more seconds remaining has a boosted utility because of the possibility of an offensive rebound. So, it is to a team's advantage to leave enough time to get the ball back, take a shot *and have a potential rebound*. But it is not quite as advantageous to get the ball back with enough time for a shot, but no possibility of a rebound.

A couple of general observations:

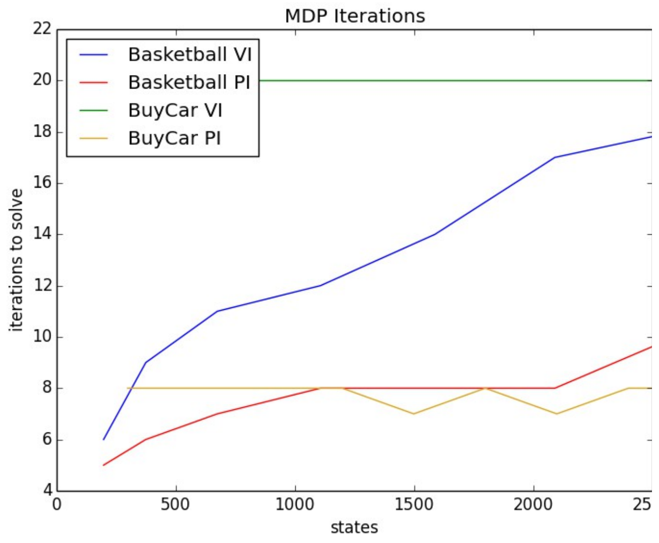
- Shooting with 10 seconds is preferable to 5 seconds because the possibility of an offensive rebound gives the offensive team a second chance, without giving the defensive team a chance.
- The model prefers 3pt shots overall.
- Rather than get a higher percentage 2pt shot and tie, the optimal policy recommends going for the win.

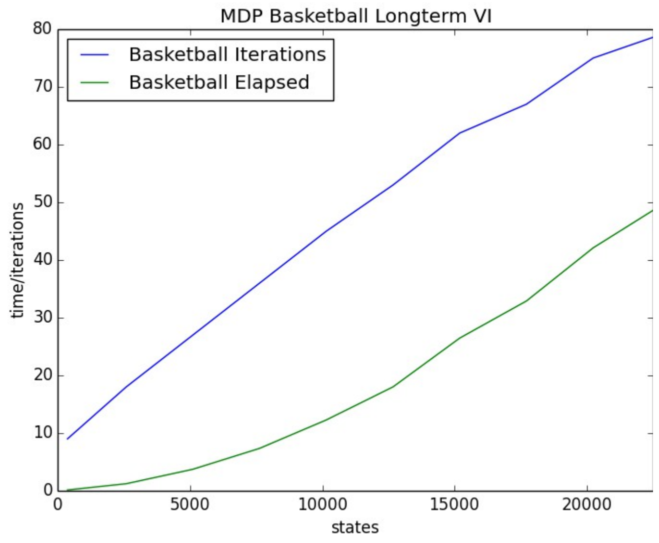
As explained above Value Iteration and Policy Iteration converge on the same utilities. However, when first comparing the generated policy graphs there are some discrepancies. Further investigation demonstrates that the policies are *trivially* different. For instance, waiting more than 5 seconds for a shot can be either the ***goto-3pt*** or ***goto-2pt*** actions, followed by going to the desired range. This difference would not exist, if, instead, the actions were ***goto-3pt-and-shoot***, ***goto-2pt-and-shoot***, and ***wait***.

Timings and Iterations

While Value Iteration and Policy Iteration arrive at the same utilities and policy (except for trivial differences), the number of iterations and time to compute is drastically different.

The graph below ("MDP Iterations") demonstrates the relationship between number of states and number of iterations needed. The first observation is that the number of iterations to solve (with either method) is very small. The 2500 states in either the Basketball or BuyCar MPD can be solved in 20 iterations. This makes sense because the models are trees – and a balanced tree would have *O(log n)* iterations. They are not balanced trees, however, so what complexity should we expect? The BuyCar MDP VI is a flat line, *O(1),* for iterations because (with a high number of states) whatever state you are in is one action from the optimal, terminal state of buying a new car. The final months have more interactions leading to the 20 iterations – but increasing states (by increasing months) doesn't change how the final months interact.

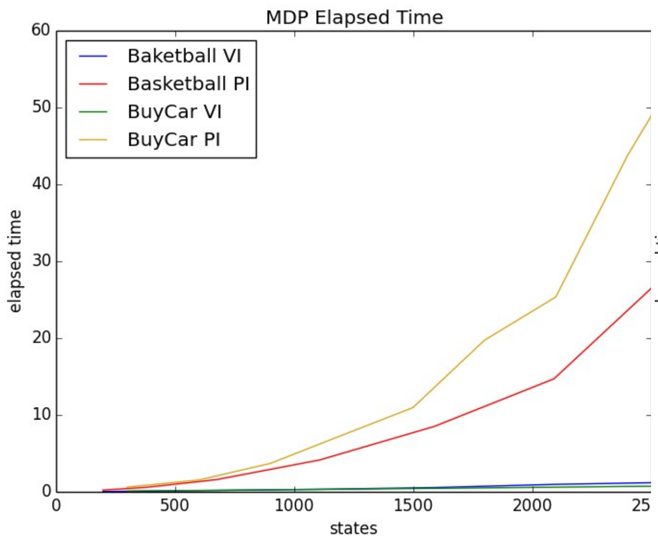


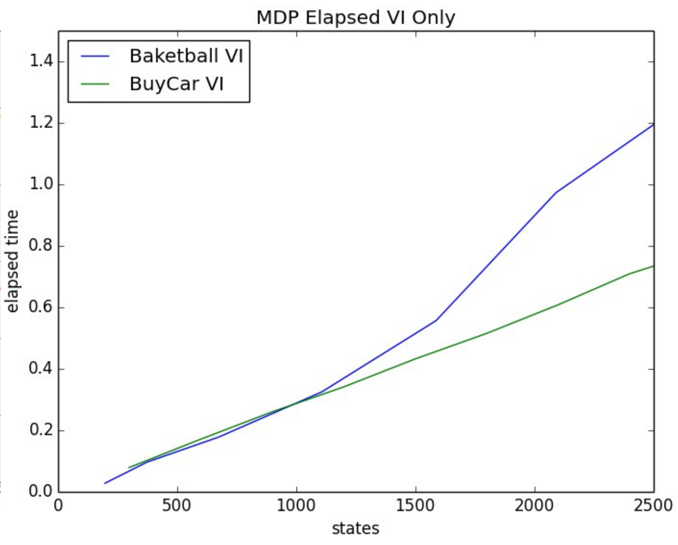


The Basketball MDP VI at first looks like it might be logarithmic in iterations ("MDP Iterations" above), but as *n* increases it looks a little more linear. For smaller *n*, the Basketball MDP resembles a balanced tree – each new level multiplies states. But after enough time, the tree reaches a maximum number of states per level. Since number of levels becomes linear with respect to number of states, the number of iterations should converge to linear complexity, *O(n).*And if we extend beyond 20000 states the relationship looks more linear ("MDP Basketball Longterm VI" above). There is still a nagging down turn after 15000 states which makes it difficult to be definitive about what will happen next.

Policy Iteration results in far fewer iterations (still "MDP Iterations" above). While we can't expect better than *O(1)* for BuyCar MDP PI, the Basketball MDP PI exhibits *O(log n)* behavior. This makes sense because each iteration solves for the interactions between **all** states (for a given policy). However, there is enough variation in policies to require an increasing number of passes with increasing number of levels in the tree (which has a linear relationship with number of states).

Elapsed time is a very different comparison ("MDP Elapsed Time" below). Basketball and BuyCar Policy Iteration are clearly greater than linear. And indeed, the generic solution of n linear equations is *O(*n^3*)* resulting in *O(*n^3 *log n).* The overhead is also quite significant, meaning that even for small *n* Value Iteration has better elapsed time.





VI elapsed time is so dwarfed by PI that we have to look at VI in isolation (“MDP Elapsed VI Only” above right). BuyCar VI is clearly linear (number of states times number of constant iterations). Basketball VI looks nearly linear but we should expect *O(n^2)* – because we evaluate each state during each iteration and the iterations are linear with respect to number of states. Looking at the long term VI top (“MDP Basketbal Longterm VI” far above), elapsed time seems it could be either *O(n^2)*, *O(n log n),* or *O(n)* . Because the linear multiplier between number of states and iterations is so small (less than 80 iterations at 25000 states), it is not surprising that we would need to look at very high states to see n^2 behavior. Unfortunately when trying to run the analysis to more than 25000 states, the algorithms hit recursion limit.

Reinforcement/Q Learning

While Value Iteration and Policy Iteration are fairly predictable for both the BuyCar and Basketball MDP's, the situation becomes more chaotic when parts of the model are unknown. To explore Reinforcement Learning we implement Q learning based on the following equation:

$$Q(s,a) \Leftarrow (1-\alpha) Q(s,a) + \alpha [R + \gamma \max_{a'} Q(s',a')]$$

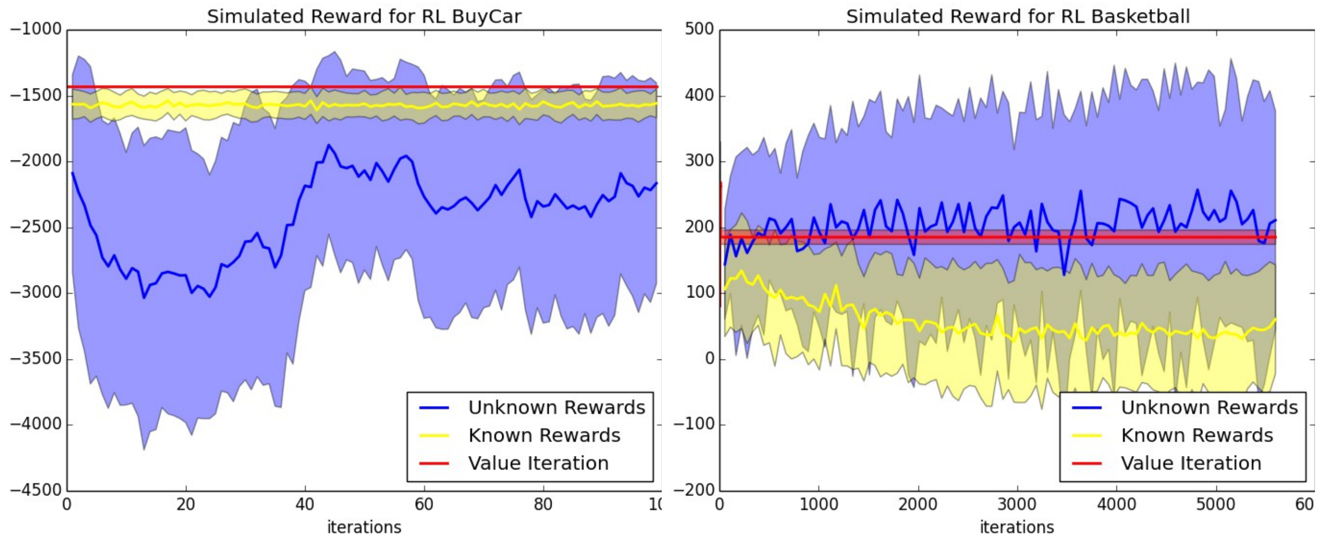
A key part of RL is deciding which action to explore. This implementation takes a random action with ε probability and the best known action otherwise. Unexplored actions start with Q = 0. ε starts at 1.0 and decays each iteration (multiply by decay rate). The decay rate is calculated so that after the total number of iterations ε will equal 0.001.

When considering the Basketball and BuyCar models, it initially made sense to implement Q-Learning with the assumption of known immediate reward and immediate transitions *before* taking an action. This makes sense for both the Basketball and BuyCar MDP's. The Basketball MDP has an immediate reward of 0 until the terminal state (where the reward of winning or losing is known). Any team playing basketball also knows the possible transitions and following states: if shoot, the team knows the resulting states are score, miss and get the offensive rebound, or miss with a defensive rebound. What is generally not known is the likelihood of each result. With the BuyCar MDP the immediate reward of buying a car is known (anyone buying a car should investigate the costs) as is the cost of a repair. And if I buy a new car, repair the old car, or keep the old car, I know for all situations what my following state possibilities are. The chance of needing a repair, however is often unknown.

Modeling Q learning in this way assigns a value of Q to each state (instead of each state-action pair). And this modeling is much closer to standard Value Iteration. We are essentially exploring our model to discover the probabilities for each transition.

But to be true to the principle of Reinforcement Learning we implement two forms of Q Learning: one with known *immediate* rewards and transitions, the other with unknown rewards and transitions (standard Q Learning).

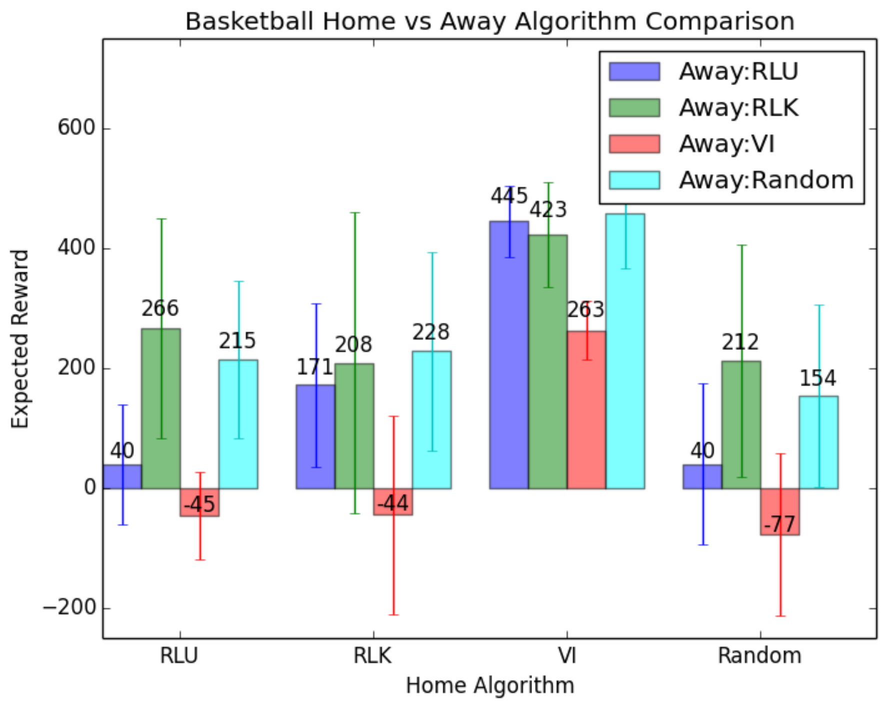
In the unknown rewards case, we simply store Q for each state-action pair and either choose randomly or the action with the best current Q value. To analyze the differences, we can sample simulations at various iteration counts. For each sample, we run the simulation 200 times to get a fair representation of expected reward. Because the learning algorithm itself can also be stochastic, we run the learning algorithm 50 times for each sample point. The following graphs show expected reward for samples with standard deviation over the 50 different runs of Q-learning.



The Q-Learning situations above are quite stochastic and show little convergence. The BuyCar MDP shows that Q-Learning with known-rewards is essentially identical to Value Iteration. Running with unknown-rewards, however, starts at -2000, becomes immediately worse, than improves until 40 iterations, after which stays around the same range. Perhaps experiments with various epsilon-decay, learning rates or gamma would enable the unknown-reward case to converge closer to our ideal.

But the Basketball case is much more interesting to explore. The most noticeable observation is that unknown-rewards Q-Learning does consistently better than known-rewards and even Value Iteration! This could indicate a problem with the implementation, but after looking into the problem deeper, these results are because the policy for *both agents* is calculated by the same algorithm. So, while unknown-rewards, may give the home team a poor policy, the same poor policy is used by the away team, resulting in a high expected reward for the home team. And known-rewards Q-Learning does worse on average but still with an expected reward greater than 0. It seems the game is skewed more toward the away team when both teams use known-rewards.

We can see a more revealing comparison in examining various combinations of home and away policies.



As before, we run 200 simulations for each algorithm combination after running Q-Learning for 5616 iterations. We run each comparison 50 times to generate a distribution with the bar representing the mean and the whiskers representing standard deviation. This analysis represents a fair degree of randomness (since each comparison is from single sample of each category). But taking 50 samples should compensate for the randomness. The standard deviation tends to separate the categories well enough to show definitive advantages.

Not surprisingly, Value Iteration is the clear winner – the baseline of optimal policy if all states, actions, transitions and probabilities are known. But if we have incomplete knowledge, knowing and modeling the transitions (RLK vs RLU) for each action seems to do slightly better. Both RL-known and RL-unknown generate policies that are better than random – although not as different as might be expected.

The results here are promising, however. The results demonstrate a clear advantage to model the game of Basketball instead of learning by trial and error. Perhaps this Basketballl MDP could be useful in practice and could improving the winning percentages of coaches and teams informed by such policies.

Conclusion

For the two time based models, Basektball and BuyCar, Value Iteration and Policy Iteration achieve the same results. Value Iteration is dramatically quicker and has simpler algorithmic complexity. Value iteration is also far simpler to implement. The optimal policies are very revealing showing providing some counter-intuitive recommendations. The recommendation to take a 3 point shot with 55 seconds left and a 2 point shot with 50 seconds left is particularly interesting. Buying a new car seems to have more long term advantages when compared to high-maintenance of an old car.

However, both these models are incomplete. A more accurate model for Basketball would represent steals, fouls (on purpose to stop the clock and unintentional), foul shooting along with real-time updated probabilities (weighting history versus the current game). Other potential states would include some degree of how good a shot is and using a smaller increment of time. Instead of modeling actions as ***goto-3pt, goto-2pt,*** and ***shoot***, a better model would be ***goto-3pt-and-shoot, goto-2pt-and-shoot,*** and ***stall***. This would allow for shots up to the last second and would reduce equivalent actions when the intent is just to use up time. Some of these changes (especially more granular time slices) might be limited in their usefulness if they result in complex strategy recommendations. For instance, "Shoot the 3 if 58 seconds with 22 on the shot clock, but 2 if 54 seconds with 22 on the shot clock, but 3 if 54 with 25 on the shot clock, etc...", would be too confusing in the time pressured end-game of basketball.

The BuyCar model also fails to capture the personal Quality of Life value of owning a new car, various distributions of repair costs, changing repair costs (for both old and new cars) and the action of leasing instead of buying.

The Reinforcement Learning experiments demonstrate that it is more difficult to optimize online systems. The experiments also showed that opponent policy makes a great difference in effectiveness of an algorithm. Future analysis could examine the roles of α (learning rate), ε (action randomness), ɣ(future discount), and ε-decay.

Credits

Matrix algorithmic complexity:
http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations
Graphviz used to generate graphs: http://www.graphviz.org/
matplotlib used to generate other graphs: http://matplotlib.org/
Python used for coding: https://www.python.org/
numpy used for matrix calculations: http://www.numpy.org/