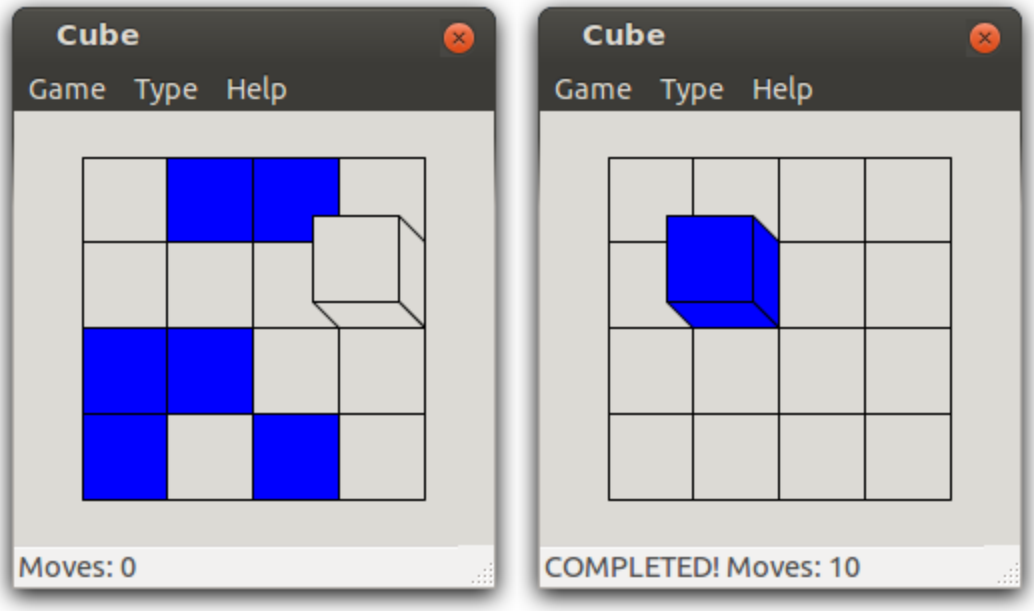
Trabajo Práctico Final

Reinforcement Learning

Kevin Allekotte

Mariano Semelman



Introducción

Para este trabajo decidimos buscar una solución de aprendizaje por refuerzos para un juego tipo puzzle llamado **Cube** (de Simon Tatham's Portable Puzzle Collection, http://www.chiark.greenend.org.uk/~sgtatham/puzzles/).
Consiste en una grilla (de tamaño variable) sobre la que se encuentran 6 cuadrados azules y un cubo, y el se juega volcando sucesivamente el cubo hacia celdas adyacentes (rodando por el tablero). Cuando se rueda sobre una celda con un cuadrado azul ("pintada"), se pega al cubo, pero si se rueda una cara pintada del cubo sobre una celda vacía, se pega a la celda (Siempre hace swap de los colores de la celda y la cara del cubo apoyada). El objetivo del juego es pintar las 6 caras del cubo :)

Estados, Acciones, Rewards, etc.

Cada estado del juego está determinado por la posición del cubo en la grilla, y la posición de cada uno de los cuadrados azules (que pueden estar en la grilla o en una de las caras del cubo). Para una grilla de 4x4, el cubo puede estar en cualquiera de las 16 celdas de la grilla, y cada uno de los 6 azules puede estar en en una de las 16 celdas ó en una de las 6 caras del cubo. El total es de $16 \times \binom{16+6}{6} = 1.193.808$ estados distintos.
Las acciones posibles son 4 (Norte, Sur, Este, Oeste), excepto cuando el cubo está en uno de los bordes, ya que no se puede salir del tablero (esta acción la consideramos ilegal o inválida y no esta modelada en el problema).
Los estados que generan reward son los estados ganadores, en el que todas las caras del cubo están azules, y que también son el End of Episode.

Solución óptima (A*)

Uno de los algoritmos que implementamos para resolver este puzzle es el de A* (A star).
Con los estados bien definidos, y sabiendo las acciones posibles desde cada uno, se puede construir el grafo que modela cualquier instancia del juego. Hay un estado inicial que representa el tablero del puzzle a resolver, y luego cada estado tiene hasta 4 vecinos, que son los estados que se obtienen luego de ejecutar alguna de las 4 acciones. La idea es encontrar un camino en este grafo, partiendo del estado inicial y llegando hasta alguno de los estados ganadores. Con A* podemos encontrar el camino más corto hasta la posición ganadora más cercana.
Empezamos por definir una heurística (cota inferior del camino mínimo) para la cantidad de pasos que faltan para llegar a la solución desde cada estado. Elegimos $h(s) = caras\ grises\ del\ cubo$, porque hacen falta al menos esa cantidad de pasos hasta que el cubo esté totalmente pintado.
$g(s)$ es la cantidad de pasos que cuesta llegar al estado s, y $f(s) = g(s) + h(s)$ es una cota inferior para la longitud total del camino pasando por s.
El algoritmo guarda una lista de estados “a explorar” (que son vecinos de estados conocidos), y explora siempre el que tenga menor $f(s)$. De esta forma recorre sucesivamente los estados que pueden llevar a una buena solución, y cuando llega a un estado ganador está seguro de que encontró el camino más corto c, porque los estados no explorados s' tienen $f(s') > c$, osea que los caminos que pasan por ellos tienen longitud de al menos c.
Al mismo tiempo se guarda el “padre” de cada estado explorado s, osea desde qué estado y con qué acción se llega a s. Con esto se puede reconstruir el camino óptimo encontrado y conocer la lista de acciones que llevan desde el estado inicial a la solución.

http://www.youtube.com/watch?v=cjCqkH9BEdc

Solución con Aprendizaje por Refuerzos

Intentar calcular la política óptima conociendo T y R de antemano dada las características propias del juego pareciera algo razonable, sin embargo no lo es. Dada la codificación de los estados podía suceder que hubiera estados no alcanzables, que las transiciones no fueran triviales, etc. Por lo que tener T y R conocidos no es factible, a corto plazo al menos. Todo esto sumado hizo que nos decidieramos por implementar Q-learning.
El algoritmo es muy similar al visto en clase, con la única diferencia que la codificación de estados cambia ligeramente, lo que repercute en que la tradicional matriz Q deja de ser una matriz estándar, y pasa a ser un diccionario. Es un diccionario ralo (sparse) que se inicializa de manera lazy, esto significa que solo crea vectores de estado para los estados que va visitando.
Por otro lado decidimos limitar los posibles movimientos a solo aquellos que tuvieran algún efecto percibible en el juego, por lo que no se le permite al agente tomar acciones como “ir a la derecha” si se encuentra en el borde derecho.
Elegimos una cantidad de episodios suficientemente grande para asegurarnos de que encuentre una solución al problema, y para cada episodio hacemos iteraciones de elegir acción y simular ambiente hasta llegar a un estado final (o abandonamos si nos lleva muchos pasos).
El método de elegir acción elige con probabilidad ϵ una acción al azar (exploración), y con probabilidad $1 - \epsilon$ una de las acciones que maximizan el reward conocido hasta el momento (que maximiza el Q, explotación).
Para simular el ambiente simplemente actualizamos las estructuras del estado actual del juego con la acción elegida.
Al final de la iteración actualizamos Q agregándole reward a tomar la acción elegida desde el estado previo en función del reward del estado nuevo, de la siguiente forma:
$Q[s][a] \ += \ \alpha \times (r \ + \ (\gamma \times max(Q[s\_next])) \ - \ Q[s][a])$,
donde α es el factor de aprendizaje, r es el reward obtenido en este paso y γ es la función de descuento.
Cuando terminamos todos los episodios esperamos haber llegado a la solución óptima en alguna de las iteraciones, y rehacemos todas las acciones que maximizan el Q de cada estado hasta llegar al estado final.

Pruebas realizadas

Probamos ver las curvas de aprendizaje para tableros de 3x3 y 4x4. En cada uno de los gráficos (Fig 1 y Fig 2) se ve como la cantidad de pasos necesarios, en los episodios que encontró solución, va decrementando a medida se realizan más episodios. Notar que la convergencia no es suave dado que un paso en falso y el agente se habrá alejado muchísimo de la solución. En las figuras se ven 4 corridas para cada caso.

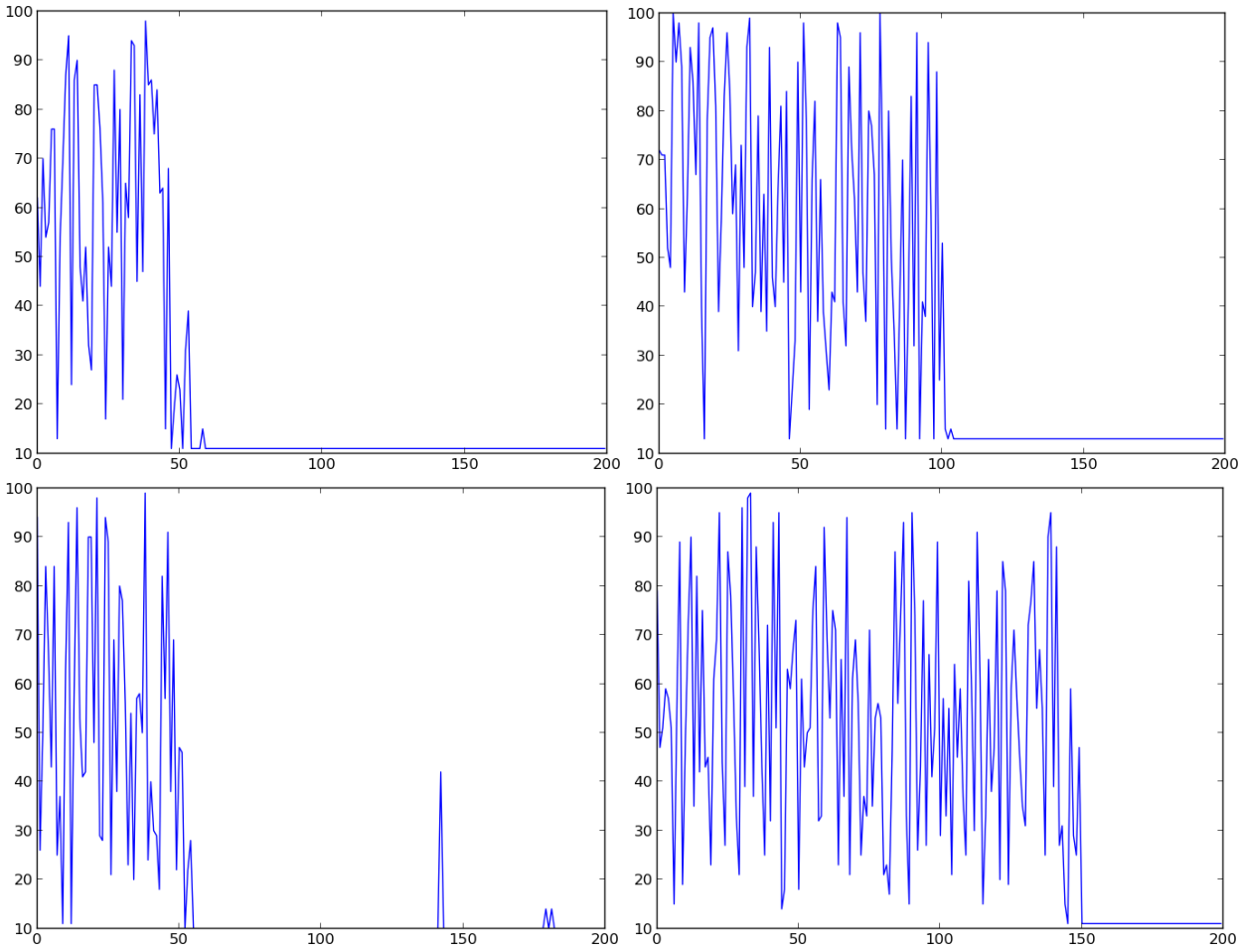


Fig 1. Tablero de 3x3, $\epsilon = 0.005$, 200 episodios

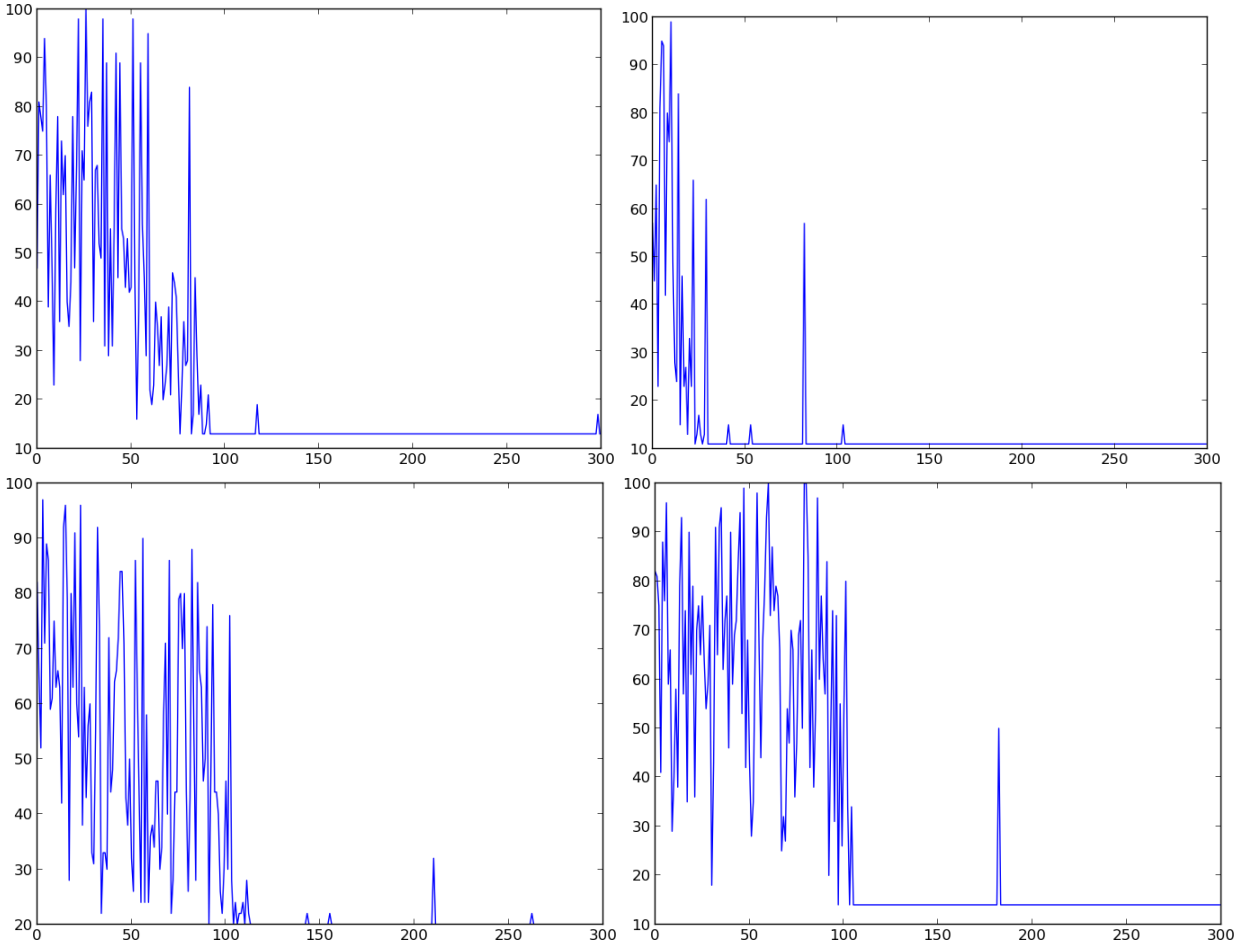


Fig 2. Tablero de 4x4, $\epsilon = 0.005$, 300 episodios

Discusión

A la hora de comparar Q-learning converge mucho más lento, por razones obvias, que A-star. Sin embargo lo interesante es que no necesitó información adicional mientras que A-star fue necesario "enseñarle" algo sobre el problema (la cantidad de caras pintadas del cubo).
En algún momento se pensó en hacer reward shaping cada vez que pintara una cara del cubo, sin embargo concluimos que sería contraproducente pues nos sesgaría las soluciones.
Analizando los resultados para *Q-learning* y sus traspies por el tipo de ambiente, creemos que usar Sarsa hubiera podido ser de utilidad. Decimos esto pues la forma en que el agente se mueve en Q-learning es muy susceptible de tomar decisiones que a corto plazo parecen correctas pero que a largo plazo son contraproducentes.
Ya a tono personal también es notable destacar que cualquier persona para resolver el juego de tamaño 3x3 le toma en promedio 20-30 movidas.

Utilización, Implementación

Está implementado en python, usando la librería **numpy** para los cálculos matriciales (especialmente `max` y `amax`), y **xdotools** para resolver el juego una vez que encontramos la solución.
Cuando inicia el programa **cube** obtenemos el identificador de la instancia del juego generado al azar en el que están codificadas las condiciones iniciales del tablero. Lo decodificamos y replicamos en nuestras propias estructuras sobre las que buscamos la solución. Una vez resuelto (con cualquiera de los métodos) le enviamos eventos de tecla a la aplicación que resuelven la instancia.

Para poder probarlo es necesario tener las siguientes librerias. Para instalarlas en Ubuntu:

```
sudo apt-get install sgt-puzzles xdotool x11-utils python-numpy
```

Para correr ejecutar las siguientes lineas:

```
./cube.py --solver="./qlearning.py" --size 3 3
./cube.py --solver="./astar.py" --size 3 3
```

Cuando arranque a correr no quitar el foco a la ventana del cubo, pués cuando termine enviará las señales de teclas para jugar y se verá la solución en pantalla.