Energy games solver

Domiziano Scarcelli - 1872664 - A.A 2023/2024

Games on Graphs

Branca della teoria dei giochi che descrive i giochi le cui regole e l'evoluzione sono rappresentate da un grafo

Arena

- Il posto in cui si svolge il gioco
- Insieme di vertici $V = V_{\text{Min}} \cup V_{\text{Max}}$
  - Finiti: arena finita
  - Infiniti: arena infinita
- Giocatore singolo o multipli. Nel nostro caso 2 giocatori: Min e Max

Play

- Token si muove da un vertice all'altro
- Il giocatore che "possiede" il vertice v muove il token attraverso un arco $v \to v'$
- La sequenza di movimenti del token denota una giocata (play) π.
  - Finita o infinita, ma non vuota

Strategie

- Funzione che mappa giocate finite in archi.
- Denotata con $\sigma : \mathrm{Paths} \to E$
- $\mathrm{Paths} = \{\pi_0, \pi_1, \ldots, .\pi_n\}$ insieme di n giocate.
- Definisce una descrizione completa del comportamento di un giocatore per ogni possibile situazione.

Condizioni

- ***Winning condition***: condizione che definisce quando un giocatore vince.
  - **Qualitativa**: è una funzione che separa le giocate vincenti $W \subset \mathrm{Paths}$ da quelle non vincenti.
  - **Quantitativa**: è una funzione che assegna un punteggio (valore reale o $\pm\infty$) ad ogni giocata.

Strategie vincenti

- In un gioco **qualitativo**, una strategia σ è vincente da un vertice v se ogni giocata che parte da v consistente con σ è vincente.
- In un gioco **quantitativo**, una strategia σ è vincente da un vertice v se per ogni giocata che parte da v consistente con σ ha un valore maggiore di una certa soglia $x \in \mathbb{R}$. Ovvero $f(\pi) \geq x$.

Energy games

Definizione del gioco e degli obiettivi

Energy Games

- Gioco quantitativo (a somma zero) a 2 giocatori (Min e Max)
- Arena: $\mathcal{A} = (G, V_{\text{Min}}, V_{\text{Max}})$
- Grafo: $G = (V, E, In, Out)$

Ogni arco $e \in E$ ha un peso $w(e) \in [-W, W]$ con $W \in \mathbb{R}$

Energia

- Peso dell'arco denota accumulo o consumo di energia:
  - Peso positivo: giocatore accumula energia
  - Peso negativo: giocatore consuma energia

Obiettivi

- Min: trovare un valore iniziale tale che possa navigare il grafo all'infinito mantenendo un'energia non negativa.
- Max: impedire l'obiettivo di Min.

Il gioco è a somma zero: se Min vince, Max perde e viceversa.

Problemi computazionali associati

Cosa possiamo calcolare

- Risolvere il gioco: booleano che indica se Min ha una strategia vincente
- Calcolare il valore del gioco: modellare funzione di valore (valore dell'energia iniziale di Min che soddisfa la condizione vincente, per ogni vertice $v \in V$.)
- Costruire una strategia ottimale per Min.

Il risolutore implementato calcola il valore del gioco.

> Notare che l'arena ha una soluzione diversa da $+\infty$ solo se non esistono cicli negativi.

Algoritmo "Value Iteration"

Trovare il valore del gioco con una complessità $O(nmW)$

Naive Value Iteration

$$\delta(l, w) = \max(l - w, 0)$$

$$\mathbb{O}^{\mathcal{G}}(\mu)(u) = \begin{cases} \min\{\delta(\mu(v), w) : u \xrightarrow{w} v \in E\} & \text{if } u \in V_{\mathrm{Min}}, \\ \max\{\delta(\mu(v), w) : u \xrightarrow{w} v \in E\} & \text{if } u \in V_{\mathrm{Max}}. \end{cases}$$

Pseudocodice

for $u \in V$ **do**
  $\mu(u) \leftarrow 0$
repeat
  $\mu \leftarrow \mathbb{O}^{\mathcal{G}}(\mu)$
until $\mu = \mathbb{O}^{\mathcal{G}}(u)$
return μ

Analisi della complessità

- Calcolare $\mathbb{O}^{\mathcal{G}}(\mu)(u)$ ha una complessità di $O(|Out(u)|)$
- Sia p la probabilità di avere un arco tra due vertici, il numero medio di archi uscenti da un vertice è $p \cdot n$.
- Il ciclo viene ripetuto per tutti gli n nodi, quindi la complessità è $O(n^2 p)$, ovvero $O(m)$.
- Il massimo numero di iterazioni prima di arrivare alla convergenza è $O(nW)$.

Quindi la complessità totale è $O(nmW)$.

Implementazione Python

```python
def _delta(self,l, w):
        return max(l-w, 0)

def _O(self, node: int):
    """
    The O^G function which returns the max value between all the
    outgoing edges from the node (if player is Max),
    or the min value (if player is Min).
    """
    values = (self._delta(self.arena.value_mapping[v], w) for
        (u, v, w) in self.arena.get_outgoing_edges(node))
    if self.arena.player_mapping[node] == Player.MAX:
        return max(values, default=0)
    else:  # player is MIN
        return min(values, default=0)

def value_iteration(self):
    """
    The naive value iteration algorithm to compute the value function.
    """
    threshold = 0.000001
    steps = 0
    max_steps = 50_000
    pbar = tqdm(total=max_steps, desc="Value iteration")

    # Maximum n iterationr, so complexity is O(n^3) in the case of edge_probability = 1
    while True:
```

Refined Value Iteration Algorithm

Come rendere l'algoritmo più efficiente?

Possiamo essere più efficienti e considerare solo un sottoinsieme di nodi per ogni iterazione.

Definiamo le seguenti variabili:

- Un insieme di vertici incorretti `Incorrecct`
- Una mappa `Count` che associa a ogni vertice di Min il numero di archi uscenti incorretti.

Definizione di "incorretto"

- Un arco $u \xrightarrow{w} v$ è incorretto se $\mu(u) < \delta(\mu(u), w)$
- Un vertice u è incorretto:
  - Ha un arco uscente incorretto e $u \in V_{\text{Max}}$
  - Tutti i suoi archi uscenti sono incorretti e $u \in V_{\text{Min}}$

function Init()
  for $u \in V$ **do**
    $\mu(u) \leftarrow 0$

  for $u \in V_{\text{Min}}$ **do**
    for $u \xrightarrow{w} v \in E$ **do**
      if $incorrect : \mu(u) < \delta(\mu(u), w)$ **then**
        $Count(u) \leftarrow Count(u) + 1$
      if $Count(u) = Degree(u)$ **then**
        Add u to Incorrect

  for $u \in V_{\text{Max}}$ **do**
    for $u \xrightarrow{w} v \in E$ **do**
      if $incorrect : \mu(u) < \delta(\mu(u), w)$ **then**
        Add u to Incorrect

function Treat(u)
$\mu(u) \leftarrow \mathbb{O}^{\mathcal{G}}(\mu)(u)$

function Update(u)

if $u \in V_{\text{Min}}$ **then**
$Count(u) \leftarrow 0$

for $v \xrightarrow{w} u \in E$ which is incorrect **do**
if $u \in V_{\text{Min}}$ **then**
$Count(v) \leftarrow Count(v) + 1$
if $Count(v) = Degree(v)$ **then**
Add v to Incorrect

```
function Main()
    Init()
    for i = 0, 1, 2, ... do
        Incorrect' ← ∅

        for u ∈ Incorrect do
            Treat(u)
            Update(u)

        if Incorrect' = ∅ then
            return μ
        else
            Incorrect ← Incorrect'
```

Analisi della complessità

- La funzione `Init` ha una complessità di $O(n + |Out(V_{\text{Min}})| + |Out(V_{\text{Max}}|))$, dove $|Out(V)|$ è il numero totale di archi uscenti da V.
- La funzione `Treat(u)` equivale a calcolare $\mathbb{O}^{\mathcal{G}}(\mu)(u)$, quindi la complessità per ogni $u \in Incorrect$ è $O(n^2p) = O(m)$, considerando il caso peggiore in cui tutti i vertici sono incorretti (quindi $|Incorrect| = n$).
- La funzione `Update(u)` ha una complessità di $O(|In(u)|)$, che equivale a $O(np)$. Quindi applicata a tutti i vertici in `Incorrect` ha una complessità di $O(n^2p) = O(m)$

Possiamo vedere che nel caso peggiore in cui $|Incorrect| = n$ la complessità è $O(nmW)$, uguale all'approccio nativo.

Empiricamente però abbiamo che $|Incorrect| < n$, quindi l'algoritmo è più efficiente.

Implementazione Python

```python
def optimized_value_iteration(self):
    incorrect: Set[int] = set()
    incorrect_prime: Set[int] = set()
    count: Dict[int, int] = {node: 0 for node in self.arena.nodes}

    def init():
        self.arena.value_mapping = {node: 0 for node in self.arena.nodes}
        pbar = tqdm(total=len(self.arena.nodes), desc="Opt Value Iteration - Init")
        # For each MIN node
        min_nodes = (n for n in self.arena.nodes
                     if self.arena.player_mapping[n] == Player.MIN)
        for node in min_nodes:
            pbar.update(1)
            for (u, v, w) in self.arena.get_outgoing_edges(node):
                if self.arena.value_mapping[u] < self._delta(self.arena.value_mapping[v], w):
                    count[u] += 1
            # If count == degree of node
            if count[node] == self.arena.get_node_degree(node):
                incorrect.add(node)

        # For each MAX node
        max_nodes = (n for n in self.arena.nodes
                     if self.arena.player_mapping[n] == Player.MAX)
        for node in max_nodes:
            pbar.update(1)
```

Generazione dell’arena

Come generare un’arena valida per il gioco

Generazione dell'arena

Parametri

- `num_nodes` : numero di nodi;
- `edge_probability` : probabilità di avere un arco tra due nodi;
- `max_weight` : definisce il range $[-W, W]$ dei pesi degli archi;

$edges \leftarrow \emptyset$
for $v \in V$ **do**
  for $u \in V$ **do**
  random_number $\leftarrow random(0, 1)$
    if random_number $\leq$ edge_probability **do**

      $w \leftarrow sample(-W, W)$
      $edge \leftarrow (v, u, w)$

      if $edge$ doesn't create a cycle **do**
        $edges \leftarrow edges \cup \{(v, u, w)\}$

Bellman Ford

- Capire se il grafo presenta cicli negativi.
- Ogni volta che viene eseguito, deve visitare tutti i nodi e tutti gli archi.
- Complessità di $O(mn)$.

```
1   def bellman_ford(self):
2       """
3       Detect negative cycles using Bellman - Ford algorithm .
4       """
5       distances = { node:0 for node in nodes }
6       # Relax edges repeatedly
7       for _ in range (len(nodes)-1):
8           for edge in edges:
9               if distances [edge[0]] + edge[2] < distances.get(edge[1],float('inf')):
10                  distances[edge[1]] = distances[edge[0]] + edge[2]
11      # Check for negative cycles
12      for edge in edges:
13          if distances[edge[0]] + edge[2] < distances.get(edge[1],float('inf')):
14      # Negative cycle found
15              return True
16      return False
```

Bellman Ford Incrementale

- Ad ogni step, un solo arco nuovo.
- Aggiornare solamente i pesi relativi al sottografo creato dall'arco.

```python
def bellman_ford_incremental(self, new_edge: Tuple[int, int, float]) -> bool:
    """
    A very efficient implementation of the Bellman-Ford algorithm
    that only checks for negative cycles related to the new edge.
    It uses the fast_edges dictionary to keep track of the edges
    and their weights, in order to avoid iterating over all the edges.
    """
    # Add the new edge
    self.edges.add(new_edge)
    self.fast_edges[new_edge[0]][new_edge[1]] = new_edge[2]

    new_distance_0 = self.distances.get(new_edge[0], 0) + new_edge[2]
    new_distance_1 = self.distances.get(new_edge[1], float('inf'))

    previous_distance_1 = self.distances.get(new_edge[1], None)

```

Complessità di $O(np)$.

Performance Evaluation

Valutare i tempi di generazione dell’arena e risoluzione del gioco con i differenti algoritmi

Risoluzione del gioco

Number of nodes	Edge probability	Time (naive)	Time (optimized)
10	0.1	2.80 ms	2.66 ms
10	0.2	3.63 ms	2.80 ms
10	0.5	2.34 ms	3.44 ms
50	0.1	3.05 ms	2.89 ms
50	0.2	3.41 ms	3.19 ms
50	0.5	4.86 ms	3.68 ms
100	0.1	3.65 ms	3.28 ms
100	0.2	4.49 ms	4.53 ms

Conclusioni

- Risolutore di energy games
- Generazione arena senza cicli negativi

Grazie per l’attenzione