Parity Games

Beau De Clercq

January 25, 2020

Abstract

In this report we will briefly discuss how it is possible to implement an easy way to solve parity games. Firstly we will describe the models that were used in this implementation, followed by a brief description of how the algorithm itself solves our game. We conclude with some examples input and output.

1 The application

1.1 Models

In this implementation two models were used: a **ParityNode** class and a **ParityGame** class.
As the name suggests, a **ParityNode** object is used to store all information pertaining to the node as described in the input file. Such an object thus has an ID, priority, owner, a list of all successors and optionally a name.
An instance of the **ParityGame** class is used to store all nodes and edges who together form a single game. It consists of a list containing all nodes V, two lists V_0 and V_1 consisting of all nodes belonging to $player_0$ or $player_1$ respectively, a list of edges E and a priority mapping Ω.

1.2 Algorithm

In our application we decided to implement the algorithm by Zielonka as described in [1, p28]. As a first step we check whether the set of vertices V is empty or not. If this is the case it means we have reached our base case and

we return two empty lists. If V is not empty, we continue with the execution by retrieving the lowest priority currently stored within the game, storing this value in m, along with the relevant nodes which we will store in M. Next we decide which player to perform the algorithm for by putting $i = m\%2$ and then determining the set of attractors $R = Attr_i(G, M)$ based on the current game along with our set of vertices M. Then we remove all nodes in R from the game, along with the according edges, followed by a recursive call, resulting in $W_i, W_j = zielonka(G \backslash R)$. If the set W_j is empty, we put $W_k = W_i \cup R$ and $W_l = \Phi$ before returning W_k, W_l as the result. If W_j is not empty, we calculate $S = Attr_{i-1}(G, W_j)$, remove all nodes in S and relevant edges from the game, make a recursive call $W_m, W_n = zielonka(G \backslash S)$ and put $W_k = W_m; W_l = W_n$ before returning W_k, W_l.
In order to determine the attractor set we run the algorithm as given in 2.2.

1.3 How to run

The application can be run in 2 ways:

- By running the run_tests script in the main directory;
- By running $pythonPG.py\{gamefile\}$ where $gamefile$ describes a parity game according to the notation described in [2, p32].

Running the application will output whether Eve (a.k.a. player 0) is able to win the game as well as generating an image of what the game looks like. The visualization requires that `graphviz` and `dot` are installed and can be found in the terminal.

2 Appendix

2.1 Zielonka

```
def zielonka(self):
if len(self.V) == 0:
  return [], []
else:
  k = len(self.V)
  m = self.min_priority()               # Retrieve lowest priority
  M = self.get_nodes_min_priority()     # Get all nodes with lowest priority
  i = m%2
  R = self.attr(i, k, M)
  self.remove_nodes(R)                  # Remove all nodes and related edges from the
       game
  Wi, Wj = self.zielonka()              # Make a recursive call, W_i' == Wi, W_(i-1)'
       == Wj
  if len(Wj) == 0:                      # If set is empty
    Wk = Wi+R                         # W_i == Wk
    Wl = []                           # W_(i-1) == Wl
  else:
    S = self.attr(i-1, k, Wj)
    self.remove_nodes(S)              # Remove all nodes and related edges from the
         game
    Wm, Wn = self.zielonka()          # Make recursive call, W_i'' == Wm, W_(i-1)''
         == Wn
    Wk = Wm                           # W_i == Wk
    Wl = Wn+S                         # W_(i-1) == Wl
  return Wk, Wl                         # return W_i, W_(i-1)
```

2.2 Attractor function

```
def attr(self, i, k, T):
if k == 0:
  return T
else:
  attrk = self.attr(i, k - 1, T)  # Attr_i^(k-1)
  #print("attrk: ", attrk)
  attrk_ids = [node.get_id() for node in attrk]
  V0 = [node.get_id() for node in self.V0]
  V1 = [node.get_id() for node in self.V1]
  edges = self.E
  edges_map = to_map(self.E)
  nodes0 = []
  nodes1 = []

  if i == 0:
    # Follow procedure for player 0
    # if there is an edge (u, v) with u in V_E where v is in attrk, add u to the
         list
    for e in edges:
      if e[1] in attrk_ids and e[1] in V0:
        nodes0.append(self.get_node(e[0]))
    # if for all edges (u, v) in V\V_E there is an edge with v in attrk, add u to
         the list
    for e in edges_map:
      if e in attrk_ids:
        for node in edges_map[e]:
          if node in V1:
            nodes0.append(self.get_node(node))
    attrs = attrk[:]
    for node in nodes0:
      if node not in attrs:
        attrs.append(node)
    return attrs

  elif i == 1:
    # Follow procedure for player 1
    # if there is an edge (u, v) with u in V\V_E where v is in attrk, add u to the
         list
    for e in edges:
      if e[1] in attrk_ids and e[1] in V1:
        nodes1.append(self.get_node(e[0]))
    # if for all edges (u, v) in V_E there is an edge with v in attrk, add u to the
        list
    for e in edges_map:
      if e in attrk_ids:
        for node in edges_map[e]:
          if node in V0:
            nodes1.append(self.get_node(node))
    attrs = attrk[:]
    for node in nodes1:
      if node not in attrs:
        attrs.append(node)
    return attrs
```

2.3 full code

```
import sys


class ParityNode:
  def __init__(self, id, priority, owner, successors, name=None):
    self.id = id
    self.priority = priority
    self.owner = owner
    self.successors = successors
    self.name = name

  def __str__(self):
    return "ParityNode: id {0}, priority {1}, owner {2}, successors {3}, name {4}"
    .format(self.id, self.priority, self.owner, self.successors, self.name)

  def get_successors(self):
    return self.successors

  def get_id(self):
    return self.id

  def get_priority(self):
    return self.priority

  def get_dot_format(self):
    label = "\""
    if self.owner == 0:
      if self.name is None:
        label += "{0} with priority {1}".format(self.id, self.priority)
        label += "\""
        return "{0} [shape=circle, label={1}]\n".format(self.id, label)
      else:
        label += "{0} with priority {1}".format(self.name.strip('"'), self.priority)
        label += "\""
        return "{0} [shape=circle, label={1}]\n".format(self.name, label)
    elif self.owner == 1:
      if self.name is None:
        label += "{0} with priority {1}".format(self.id, self.priority)
        label += "\""
        return "{0} [shape=diamond, label={1}]\n".format(self.id, label)
      else:
        label += "{0} with priority {1}".format(self.name.strip('"'), self.priority)
        label += "\""
        return "{0} [shape=diamond, label={1}]\n".format(self.name, label)

  def get_dot_name(self):
  if self.name is None:
    return self.id
  else:
    return self.name


class ParityGame:
  def __init__(self):
    self.V = []          # all nodes of the game
    self.V0 = []         # all nodes belonging to player 0
    self.V1 = []         # all nodes belonging to player 1
    self.E = []          # all edges between the nodes
    self.omega = {}      # maps the priority of a node to the ID of the node
```

```python
def add_node(self, id, priority, owner, successors, name=None):
  node = ParityNode(id, priority, owner, successors, name)
  self.V.append(node)
  self.omega[str(id)] = priority
  if owner == 1:
    # add to V1
    self.V1.append(node)
  elif owner == 0:
    # add to V0
    self.V0.append(node)

def add_edges(self):
  # Loop over all nodes, add edge between a node and its successors
  for node in self.V:
    successors = node.get_successors()
  for succ in successors:
    edge = node.get_id(), succ
    self.E.append(edge)

def get_states_player_1(self):
  return self.V1

def get_states_player_0(self):
  return self.V0

def get_states(self):
  return self.V

def get_edges(self):
  return self.E

def get_node(self, id):
  for node in self.V:
    if node.get_id() == id:
      return node

def get_edges_v0(self):
  v_ids = [node.get_id() for node in self.V0]
  edges = []
  for edge in self.E:
    if edge[0] in v_ids:
    edges.append(edge)
  return edges

def get_edges_v1(self):
  v_ids = [node.get_id() for node in self.V1]
  edges = []
  for edge in self.E:
    if edge[0] in v_ids:
      edges.append(edge)
  return edges

def min_priority(self):
  prio = self.V[0].get_priority()
  for node in self.V:
    n_prio = node.get_priority()
    if n_prio < prio:
      prio = n_prio
  return prio

def max_priority(self):
  prio = self.V[0].get_priority()
  for node in self.V:
    n_prio = node.get_priority()
    if n_prio > prio:
      prio = n_prio
  return prio

def get_nodes_min_priority(self):
  min_prio = self.min_priority()
  nodes = []
  for node in self.V:
    if node.get_priority() == min_prio:
      nodes.append(node)
  return nodes
```

```
def get_nodes_max_priority(self):
  min_prio = self.max_priority()
  nodes = []
  for node in self.V:
    if node.get_priority() == min_prio:
      nodes.append(node)
  return nodes

def attr(self, i, k, T):
  if k == 0:
    return T
  else:
    attrk = self.attr(i, k - 1, T)  # Attr_i^(k-1)
    attrk_ids = [node.get_id() for node in attrk]
    V0 = [node.get_id() for node in self.V0]
    V1 = [node.get_id() for node in self.V1]
    edges = self.E
    edges_map = to_map(self.E)
    nodes0 = []
    nodes1 = []

    if i == 0:
      # Follow procedure for player 0
      # if there is an edge (u, v) with u in V_E where v is in attrk,
      # add u to the list
      for e in edges:
        if e[1] in attrk_ids and e[1] in V0:
          nodes0.append(self.get_node(e[0]))
          # if for all edges (u, v) in V\V_E there is an
          # edge with v in attrk, add u to the list
      for e in edges_map:
        if e in attrk_ids:
          for node in edges_map[e]:
            if node in V1:
              nodes0.append(self.get_node(node))
      attrs = attrk[:]
      for node in nodes0:
      if node not in attrs:
      attrs.append(node)
      return attrs

    elif i == 1:
      # Follow procedure for player 1
      # if there is an edge (u, v) with u in V\V_E where v is in attrk,
      # add u to the list
      for e in edges:
        if e[1] in attrk_ids and e[1] in V1:
          nodes1.append(self.get_node(e[0]))
      # if for all edges (u, v) in V_E there is an edge with v in attrk,
      # add u to the list
      for e in edges_map:
        if e in attrk_ids:
          for node in edges_map[e]:
            if node in V0:
              nodes1.append(self.get_node(node))
      attrs = attrk[:]
      for node in nodes1:
      if node not in attrs:
      attrs.append(node)
      return attrs

def remove_nodes(self, nodes):
  #remove edges
  for e in self.E:
    if self.get_node(e[0]) in nodes or self.get_node(e[1]) in nodes:
      self.E.remove(e)
  #remove nodes
  self.V = [node for node in self.V if node not in nodes]
  self.V0 = [node for node in self.V0 if node not in nodes]
  self.V1 = [node for node in self.V1 if node not in nodes]

def zielonka(self):
  if len(self.V) == 0:
    return [], []
  else:
    k = len(self.V)
    m = self.min_priority()             # Retrieve lowest priority
```

```
      M = self.get_nodes_min_priority()    # Get all nodes with lowest priority
      i = m%2
      R = self.attr(i, k, M)
      self.remove_nodes(R)                 # Remove all nodes and related edges from the
          game
      Wi, Wj = self.zielonka()             # Make a recursive call, W_i' == Wi, W_(i-1)'
          == Wj
      if len(Wj) == 0:                     # If set is empty
        Wk = Wi+R                        # W_i == Wk
        Wl = []                          # W_(i-1) == Wl
      else:
        S = self.attr(i-1, k, Wj)
        self.remove_nodes(S)             # Remove all nodes and related edges from the
            game
        Wm, Wn = self.zielonka()         # Make recursive call, W_i" == Wm, W_(i-1)" ==
            Wn
        Wk = Wm                          # W_i == Wk
        Wl = Wn+S                        # W_(i-1) == Wl
      return Wk, Wl                        # return W_i, W_(i-1)

  def print_game_result(self, result):
    if len(result[0]) > 0:
      print("Eve can win")
    else:
      print("Eve cannot win")
    return

  def toDot(self, name):
    file = open(name, "w+")
    file.truncate(0)
    file.write("digraph {\n//graph [rankdir=LR]\n")
    for node in self.V:
    file.write(node.get_dot_format())
    for edge in self.E:
      n0 = self.get_node(edge[0])
      n1 = self.get_node(edge[1])
      file.write("{0} -> {1};\n".format(n0.get_dot_name(), n1.get_dot_name()))
    file.write("}")
    file.close()


def to_map(edges):
  ''' Transform edges (u, v) from pairs to a map with key v '''
  map = {}
  for e in edges:
    if e[1] not in map.keys():
      map[e[1]] = [e[0]]
    else:
      map[e[1]].append(e[0])
return map

def main(PGFile):
  game = ParityGame()
  f= open(PGFile)
  lines = f.readlines()
  parity = -1
  #print(len(lines))
  if len(lines[0].split(' ')) == 2:
    parity = int(lines[0].split(' ')[1].split(';')[0])
    print("Optional header found: parity = {0}".format(parity))
  if parity != -1:
    for i in range(1, len(lines)):
      parts = lines[i].split(' ')
      id = int(parts[0])
      priority = int(parts[1])
      owner = int(parts[2])
      successors = []
      s = parts[3].split(',')
      for succ in s:
        successors.append(int(succ.strip(';\n')))
      if len(parts) == 5:
        game.add_node(id, priority, owner, successors, parts[4].strip(';\n'))
      else:
        game.add_node(id, priority, owner, successors)
    game.add_edges()
  elif parity == -1:
    for i in range(0, len(lines)):
      parts = lines[i].split(' ')
      id = int(parts[0])
      priority = int(parts[1])
      owner = int(parts[2])
      successors = []
      s = parts[3].split(',')
      for succ in s:
        successors.append(int(succ.strip(';\n')))
      if len(parts) == 5:
        game.add_node(id, priority, owner, successors, parts[4].strip(';\n'))
      else:
        game.add_node(id, priority, owner, successors)
    game.add_edges()
  game.toDot("{0}.dot".format(PGFile.split('.')[0]))
  result = game.zielonka()
  game.print_game_result(result)


if __name__== "__main__":
  PGFile = str(sys.argv[1])
  main(PGFile)
```

References

[1] Guillerma A. Pérez, *PLAYING GAMES TO SYNTHESIZE REACTIVE SYSTEMS*

[2] Oliver Friedmann, Martin Lange, *The PGSolver Collection of Parity Game Solvers*