به نام خدا

پاسخ تمارین سری دوم

حسام مومیوند فرد

۸۱۰۸۰۳۰۶۳

فهرست

تعریف شرایط سه گانه

Safety

این شرط به دو بخش تقسیم میشود: agreement , validity

Agreement

به این معنا است که پس از اتمام و توقف برنامه و تصمیم گیری ، هیچ دو پردازشی تصمیم متفاوت نگرفته باشند. برای این کار نیاز داریم که تصمیم تمام node ها را پس از اتمام برنامه بدانیم و با مقایسه‌ی تصمیم ان‌ها این شرط را بررسی کنیم

Validity

این شرط به آن معناست که اگر

1. اگر تمام node ها تصمیم اولیه خود را 0 در نظر گرفته باشند ، پس از اتمام الگوریتم کماکان باید تصمیم آن ها 0 باشد.
2. اگر تمام node ها تصمیم اولیه خود را 1 در نظر گرفته باشند، پس از اتمام اجرای الگوریتم و در صورت گم نشدن پیام کماکان تصمیم تمام آن ها باید 1 باشد.
3. در صورتی که تصمیم تمام node ها در ابتدای کار یکسان نباشد این شرط تعریف نشده است.

Liveness

Termination

این شرط بیان گر این است که الگوریتم حتما پایان می‌یابد. در این مسئله شرط پایان پذیری ساده است زیرا ما فقط به تعداد محدودی راند الگوریتم را اجرا میکنیم.

Fault tolerance

این شرط بیانگر این است که سیستم ما در برابر خطا (در این الگوریتم خطای لینک – گم شدن پیام) مقاوم باشد. یعنی در صورت بروز خطا تا حدی (قابل محاسبه) خطا را تحمل کند و جواب درست به ما بازگرداند.

- در انتهای تمرین به بررسی ارضا شدن و یا نشدن این شروط خواهیم پرداخت.

فرضیات

توضیح مسئله

node

در این مسئله ما فرض میکنیم یک تعداد node در ابتدا تولید می‌شوند و هرکدام یک value به معنای تصمیم اولیه دارند.

در ادامه این node ها به تبادل پیام با یکدیگر میپردازند و پس از یک تعداد راند مشخص (هر راند شامل یک گروه از تبادل پیام‌ها است) تصمیم گیری میکنند که آیا حمله‌ای صورت بگیرد یا خیر. (حمله برگرفته از مسئله ژنرال ها است و در مثال های توزیع شده بیشتر منظور commit کردن و یا نکردن است)

message

در این میان برخی از پیام‌ها در شبکه دراپ می‌شوند و به دست گیرنده نمیرسند. (فرض ما بر این است که نرخ دراپ شدن پیام ها را می‌دانیم.)

از طرفی هرپیام شامل محتوایی به صورت زیر خواهد بود:

- بردار value که هر درایه آن برابر با تصمیم اولیه مربوط به node شماره ایندکس است. در ابتدا هر نود تنها تصمیم خود را میداند و از تصمیم دیگر نود ها هیچ اطلاعی ندارد پس به جز درایه مربوط به ایندکس خود نود مابقی درایه ها مقدار None میگیرند.
- بردار information_level که هر درایه آن مربوط به سطح اطلاعات (ورژن اطلاعات) ای است که از دیگر نود ها در باره‌ی تصمیم آن ها میداند. درایه های این بردار در ابتدا برابر با منفی 1 هستند زیرا از تصمیم هیچ نودی اطلاعی نداریم اما به مرور زمان تکمیل می‌شوند. درایه مربوط به خود نود برابر با 0 است.
- متغیر کلید. این متغیر توسط نود شماره 1 انتخاب میشود و یک مقدار صحیح رندوم در بازه‌ی [1,r] است. در ابتدا فقط نود شماره یک از مقدار کلید مطلع است (خودش آن را انتخاب کرده) اما به مرور زمان با انتقال پیام ها دیگر نود ها نیز از مقدار کلید آگاه خواهند شد.

state

پس از هر راند و تبادل پیام ها هر نود از یک state به state جدیدی منتقل میشود. این state ها با مقادیری مشخص می‌شوند که در ادامه به آن ها خواهیم پرداخت

- بردار value که مشابه با بردار value پیام ها است با این تفاوت که بعد از هربار دریافت یک پیام بردار value داخلی نود در آن state به این شکل آپدیت خواهد شد که: در صورتی که ایندکسی از بردار داخلی برابر None بود اما در درایه مربوطه در پیام حاوی مقدار بود، درایه مربوط به بردار داخلی مقدار خود را از درایه مشابه در پیام میگیرد.

- بردار level که مقدار آن با دریافت هر پیام به این شکل آپدیت میشود که: به ازای تک تک ایندکس های هر دو بردار داخلی و بردار پیام ، مقدار بردار داخلی برابر با max مقدار بین دو درایه هم ایندکس در دو بردار است به جز ایندکس مربوط به شماره خود node که پس از بروز رسانی بردار داخلی با max گرفتن، مقدار درایه مربوط به ایندکس خود نود برابر با min مقدار بین تمام درایه ها به علاوه یک واحد خواهد بود.
- مقدار کلید. در صورتی که نود شماره 1 نباشیم مقدار کلید را نمیدانیم و با دریافت هر پیام چک میکنیم که اگر مقدار کلید مندرج در پیام برابر با مقداری به جز unknow بود آن را در کلید خود ذخیره میکنیم.

Decision making

تصمیم گیری به این معناست: در زمانی که تعداد راند های اجرای الگوریتم برابر با مقدار r (بازه ای که کلید را از آن انتخاب کردیم) شد ما محکوم به تصمیم گیری خواهیم بود

روال تصمیم گیری به این شکل است:

در صورتی که ما مقدار کلید را بدانیم و information_level مربوط به ایندکس خودمان باز کلید بزرگتر یا مساوی باشد و مقدار بردار value داخلی ما به ازای تمام درایه ها برابر با یک باشد آنگاه تصمیم ما یک خواهد بود (حمله کردن یا commit کردن) در غیر این صورت تصمیم صفر است.

لیست فرضیات

- فرض میکنیم که تعداد کل نود ها را از ابتدای مسئله میدانیم و یک عدد فیکس و ثابت است که در طول مسئله تغییری نمیکند
- فرض میکنیم که تعداد پیام های گمشده در شبکه را میدانیم و میدانیم که کدام پیام به کدام نود نرسیده است
- فرض میکنیم هیچ یک از نود ها دروغگو نیستند (محتوای پیام ها قابل اطمینان است)
- فرض میکنیم شبکه محتوای پیام ها را تغییر نمیدهد و صرفا برخی از پیام ها را نمیفرستد
- فرض میکنیم نود ها در طول مراحل اجرای الگوریتم تصمیم خودشان را تغییر نمیدهند
- فرض میکنیم که نود های ما در یک گراف کامل به یکدیگر متصلند و هر نود با تمام نود های دیگر ارتباط دارد
- فرض میکنیم که یک نود پیغامی را برای خودش ارسال نمیکند.
- فرض میکنیم نود ها در هر راندی توانایی تصمیم گیری برای حمله کردن و یا نکردن را دارند اما تنها در راند آخر r ام تصمیم آن ها را بررسی میکنیم.

توضیحات (code)

کد ما شامل یک کلاس و دو فایل است

Class node

این فایل به تعریف کلاس نود میپردازد. ورودی کلاس ما مقادیر:

- Node_id: ای دی نودی که میخواهیم بسازیم
- Initial_value: مقدار تصمیم اولیه
- R: بازه ای که کلید در آن قرار دارد
- Number_of_nodes: تعداد کل نودهایی که میخواهیم بسازیم

هستند.

تابع init

در این تابع ما به تعریف مقادیر کلاس میپردازیم و به آن ها مقدار میدهیم.

```
def __init__(self, node_id, initial_value, r, number_of_nodes):
  self.id = node_id
  self.initial_value = initial_value
  self.r = r
  self.number_of_nodes = number_of_nodes

  self.key = None
  self.choose_key()  # choose the key value

  self.value = [None] * self.number_of_nodes # values
  self.value[node_id] = initial_value # set initial value
  self.level = [-1] * self.number_of_nodes # levels
  self.level[self.id] = 0 # set initial level
```

شاید تنها بخش قابل توضیح این کد تابع SELF.CHOOSE است که در آن ما مقدار کلید را انتخاب میکنیم.

تابع choose_key

در این تابع ما چک میکنیم که آی دی نود ما برابر یک است یا خیر در صورتی که یک باشد، یک عدد رندوم در بازه [1,r] به عنوان کلید انتخاب میکنیم و در صورتی که آی دی یک نباشد مقدار none را به کلید اختصاص میدهیم.

```
def choose_key(self):
  self.key = randint(1, self.r) if self.id == 1 else None
```

تابع receive_message

این تابع نحوه بخورد نود را با پیغام دریافتی مشخص میکند. ابتدا بخش های مختلف پیام را از یکدیگر تفکیک کرده و هر بخش را به تابع مربوطه پاس میدهیم. ورودی این تابع یک پیغام است.

```
def receive_message(self, message):
  message_value = message["value"]
  message_level = message["information_level"]
  self.key = message["key"] if self.key is None else self.key # updating key; if my key is none then
                                                              # i should update my key
  self.update_value(message_value) # updating the value vector
  self.update_level(message_level) # updating the information level vector
```

تابع update_level

ورودی این تابع بخش level از پیغام دریافتی است

```
def update_level(self, message_level):
  for i in range(self.number_of_nodes):
    if i != self.id:
      self.level[i] =max(self.level[i], message_level[i])
  min_value = min([value for i, value in enumerate(self.level) if i != self.id])
  self.level[self.id] = min_value + 1
```

و در ادامه با توجه به تعریفی که از نحوه آپدیت کردن مقدار level در هر state داشتیم، به آپدیت کردن میپردازیم.

تابع update_value

این تابع هم بخش value از پیغام دریافتی رو به عنوان ورودی میگیرد و با توجه به نحوه آپدیت کردن state که در صورت مسئله داده شده به آپدیت کردن value در state مورد نظر میپردازیم

```
def update_value(self, message_value):
  for i in range(self.number_of_nodes):
    if self.value[i] is None and message_value[i] is None:
      continue
    if self.value[i] is None and message_value[i] is not None:
      self.value[i] = message_value[i]
    if self.value[i] is not None and message_value[i] is None:
      continue
```

```
    if self.value[i] is not None and message_value[i] is not None:
      self.value[i] = max(message_value[i], self.value[i])
```

تابع generate_message

این تابع یکی از حیاتی ترین توابع کلاس ما است که در آن پیامی که در راند بعدی میخواهیم به دیگر نود ها ارسال کنیم را میسازیم. خروجی این تابع یک پیغام خواهد بود.

```
def generate_message(self):
  message = {"information_level": self.level,
        "value": self.value,
        "key": self.key}
  return message
```

تابع decision_making

در این تابع که در راند r ام فراخوانی میشود، نود به تصمیم میگیرد که حمله کند یا خیر.

ورودی این تابع مقدار round و خروجی آن یک مقدار بولین است. در صورتی که 1 بازگرداند تصمیم بر حمله است و 0 به معنای تصمیم برای عدم حمله خواهد بود.

با توجه به نحوه پیاده سازی این تابع نود ها در هر راندی میتوانند برای حمله کردن یا نکردن تصمیم گیری کنند زیرا ما مقدار راند را به این تابع به عنوان ورودی میدهیم اما نحوه پیاده سازی از تصمیم گیری در هر راندی به جز راند r ام جلوگیری میکند.

```
def decision_making(self, round):
  if self.r == round:
    if self.key is not None and all(v == 1 for v in self.value) and self.level[self.id] >= self.key:
      return 1
    else:
      return 0
```

تابع repr

این تابع برای نمایش نود در صورتی است که ما نود را با تابع print() ببینیم.

```
def __repr__(self):
  return f"{"value = ", self.value}{"information level = ", self.level}{"key = ", self.key}"
```

این تابع در حل مسئله ما نقشی ندارد و صرفا برای دیباگ بهتر پیاده سازی شده است.

Main

در این فایل هدف ما ساختن تعدادی نود، شبیه سازی ارسال پیام و اپدیت کردن استیت ها، تکرار این شبیه سازی، محاسبه‌ی validity , agreement است.

تابع main

```
if __name__ == "__main__":
    r = int(input("Enter number of rounds : "))
    number_of_nodes = int(input("Enter number of nodes : "))
    number_of_simulations = int(input("Enter number of simulations : "))
    success_messages_ratio = int(input("Enter success messages ratio (your number / 100) : "))
    zero_decision_ratio = int(input("Enter the ratio of initial 1 decisions (your number / 100) : "))

    output_data = [] # data to show in table
    false_validity_counter = 0 # global variable to count the number of validity falses in all of the simulations
    false_agreement_counter = 0 # like validity counter but for agreement
    for i in range(number_of_simulations): # start simulation
        value = [] # value of all nodes
        nodes = generate_nodes() # generate all nodes and append all nodes initial value into the value[]
        drop_message_flag = False # flag to figure out is any message dropped or not ( uses in validity )
        message_passing_simulation() # simulate massages pass through nodes
        decisions = [] # decisions made by nodes after massages passed
        result() # append all nodes decisions into decisions[]
        validity = check_validity() # validity checker
        agreement = check_agreement() # agreement checker

        # create sample output and add this loop of simulation at the end of simulation's datas
        sample_output = (i, value, decisions, agreement, validity) # create tuple to append output_data
        output_data.append(sample_output) # append tuple

    agreement_percentage, validity_percentage = calculate_validity_agreement_per_repeats()
    # the last line of data should be percentages
    last_line_sample = ("-", "-", "-", agreement_percentage, validity_percentage) # create last line ( percentages )
    output_data.append(last_line_sample) # append last line to output date
    create_table() # function to create and print the table
```

در این تابع ما ابتدا از کاربر ورودی های زیر را میگیریم:

1. R: مقدار بازه ای که کلید از آن انتخاب میشود
2. Number_of_nodes: تعداد کل نود هایی که میخواهید ساخته شوند
3. Number_of_simulations: تعداد دفعاتی که میخواهید با مقادیر اولیه رندوم و با تعداد نودهای خواسته شده در بازه کلید داده شده شبیه سازی انجام شود.

4. success_message_ratio: نرخ احتمال خطای انتقال پیام شبکه (به درصد ، 100 درصد برابر با این است که هیچ پیامی گم نشود و 0 درصد بیانگر این است که شبکه هیچ پیامی را منتقل نمیکند)
5. zero_initial_value_ratio: بیانگر احتمال مقدار صفرهایی است که در تصمیم اولیه نودها هستند.

سپس متغیر output_data را میسازیم که نتیجه هربار شبیه سازی را به آن اضافه کنیم تا در نهایت در جدول نمایش دهیم

در ادامه دو شمارنده میسازیم که با استفاده از آن ها تعداد validity , agreement هایی که در طول شبیه سازی false شده اند را نگه داری کنیم. در حلقه for شبیه سازی را آغاز میکنیم:

در گام اول بردار value مربوط به تمام node ها را تشکیل میدهیم.

نود ها را میسازیم

متغیر drop_message_flag برای این است که اگر پیامی در شبکه دراپ شد متوجه آن شویم زیرا در زمان تست validity شرطی وابسته به اینکه پیامی در شبکه دراپ شده است یا خیر وجود دارد.

تابع message_passing_simulation مراحل شبیه سازی ارسال پیام را انجام می‌دهد. در این تابع پیام های هر نود ساخته میشوند و در یک لیست ذخیره می‌شوند. سپس با فراخوانی یک تابع دیگر تعدادی از این پیام ها به صورت رندوم اما با نرخ دراپ وارد شده توسط کاربر حذف میشوند و سپس هر پیام به گیرنده خود تحویل داده میشود. این عملیات درون تابع به تعداد r راند انجام میشود.

بعد از تحویل دادن پیام ها نوبت تصمیم گیری فرا میرسد. متغیر decision برای ثبت تصمیم تمام نود ها ساخته شده است.

سپس تابع result را صدا میزنیم و نتایج را با استفاده از این تابع بدست آورده و ثبت میکنیم. در ادامه مقدار validity , agreement را محاسبه کرده و ذخیره میکنیم.

بعد از هر دور اجرا شدن یک سطر سمپل برای ثبت در دیتاهای خروجی میسازیم و در دیتای خروجی اضافه میکنیم.

پس از اتمام حلقه simulation؛ نرخ خطاهای validity , agreement را محاسبه کرده و حاصل را در آخرین سطر دیتای خروجی اضافه میکنیم.

در نهایت تابع create_table را صدا زده تا داده های خروجی را با فرمت خواسته شده نمایش دهد.

تابع generate_nodes

در این تابع ما نود ها را میسازیم و آن ها را به یک لیست اضافه میکنیم. در انتهای تابع ما یک لیست از نودهای ساخته شده خواهیم داشت. که مقادیر اولیه آن ها به صورت احتمالی با نرخ داده شده توسط کاربر تولید می‌شوند.

```
def generate_nodes():
  list_of_nodes = []
  global value
  value = generate_random_array(number_of_nodes, zero_initial_value_ratio)
  for node_id in range(number_of_nodes):
    list_of_nodes.append(Node(node_id=node_id, initial_value=value[node_id],
number_of_nodes=number_of_nodes, r=r))
return list_of_nodes
```

تابع generate_messages

در این تابع هدف ما ساختن پیامی است که هر نود ارسال میکند. حائز اهمیت است که ما پیام ها را ارسال نمیکنیم بلکه در این تابع اگر یک نود بخواهد پیامی ارسال کند، ما آن پیام را میسازیم. ارسال کردن پیام و تعیین میزان خطای شبکه در توابع دیگر خواهد بود. در انتهای تابع ما تمام پیام ها را در یک لیست از پیام ها ذخیره میکنیم به طوری که هر درایه از این لیست پیامی را شامل میشود که نود مربوط به آن ایندکس میخواهد ارسال کند.

```
def generate_messages(): # all nodes generate the message
  messages = []
  for i in range(len(nodes)):
    messages.append(nodes[i].generate_message())
  return messages
```

تابع generate_random_array

این تابع وظیفه دارد تا یک آرایه با مقادیر رندوم با تولید کند. منظور از رندوم این است که نرخ مقادیر صفر و یک در این تابع احتمالاتی است به این معنا که ما اعداد رندوم در بازه 0 تا 100 تولید میکنیم و با توجه به ورودی کاربر اگر عدد رندوم ما از ورودی بزرگتر بود مقدار 1 و در غیر این صورت مقدار صفر را در هر ایندکس قرار میدهیم.

ورودی های این تابع طول آرایه و عدد ترشولد هستند و خروجی تابع یک آرایه از اعداد صفر و یک میباشد

```
def generate_random_array(length, zero_ratio):
  random_array = []
  for index in range(length):
    random_variable = randint(0, 100)
```

```
    random_array.append(0) if random_variable > zero_ratio else random_array.append(1)
  return random_array
```

تابع deliver_messages

در این تابع ما پیام های تولید شده را به نود ها تحویل میدهیم و نود ها بعد از دریافت پیام به پروسه بعد از دریافت خود خواهند پرداخت.

```
def deliver_messages(messages):
  for receiver_node_id in range(number_of_nodes):
    delivery_list = generate_random_array(number_of_nodes, success_message_ratio)
    global drop_message_flag
    for message_index in range(len(messages)):
      for delivery_list_index in range(len(delivery_list)):
        if message_index != delivery_list_index and delivery_list[delivery_list_index] == 1:
          nodes[receiver_node_id].receive_message(messages[message_index])
        elif delivery_list[delivery_list_index] == 0:
          drop_message_flag = True
```

در این تابع ما ابتدا برای هر نود یک لیست تحویل (تصادفی) میسازیم. سپس با دو حلقه تو در تو کل پیام ها و کل لیست تحویل را پیمایش کرده هرجا که مقدار لیست تحویل یک بود پیام مربوطه را به نود تحویل میدهیم و نود مقادیر خود را با توجه به پیام آپدیت میکند. در صورتی که در پیمایش لیست تحویل به مقدار صفر برخورد کنیم نیز پرچم دراپ شدن پیام را بالا میبریم تا در ادامه در زمان محاسبه validity از آن استفاده کنیم.

تابع message_passing_simulation

این تابع محل اتصال و نقطه مشترک چندین تابع است. ما در این تابع به تعداد راند های داده شده شبیه سازی را اجرا میکنیم که شامل تولید پیام ، تولید لیست تحویل و تحویل دادن پیام ها است.

```
def message_passing_simulation():
  for round in range(r):
    messages = generate_messages()
    delivery_list = generate_delivery_list()
    deliver_messages(messages, delivery_list)
```

تابع result

در این تابع ما نتیجه گیری را بعد از اتمام شبیه سازی انجام میدهیم. هر نود تابع تصمیم گیری خودش را صدا زده و تصمیمات نود ها را در یک لیست ذخیره میکنیم.

```
def decision_making():
  for node_index in range(number_of_nodes):
    decisions.append(nodes[node_index].decision_making(r))
```

تابع check_validity

در این تابع ما validity را با توجه به شروط داده شده در صورت مسئله چک میکنیم.

```
def check_validity():
  global false_validity_counter
  if ((not(all(not v for v in value)) or all(not d for d in decisions)) or
      (not(all(value) and not drop_message_flag) or all(decisions)) or
      (any(value) and not all(value))):
    return True
  else:
    false_validity_counter += 1
    return False
```

چک کردن validity کاملا منطبق بر نوشته های سودو کد و کتاب است با این تفاوت که در حالت آغاز نود ها با مقدار اولیه های متفاوت validity همواره یک خواهد بود یا به عبارتی تعریف نشده است زیرا مقدار آن تنها برای حالاتی تعریف میشد که تمام نود ها با مقدار برابر 0 و یا برابر 1 کار خود را آغاز کنند. همچنین در توضیحات کتاب از گزاره‌ی اگر تمام نود ها با مقادیر اولیه یک شروع کنند و پیامی گم نشود آنگاه باید تمام تصمیمات برابر یک باشند استفاده شده است که با استفاده از هم ارزی های منطقی ما این گزاره را به نقیض ،تمام مقادیر اولیه یک باشند و پیامی گم نشود، یا تمام تصمیمات یک باشند تبدیل کردیم. همچنین در توضیحات آمده است که اگر تما مقادیر با صفر شروع کردند آنگاه خروجی باید صفر باشد و این گزاره را نیز با استفاده از نقیض، تمام نود ها با صفر شروع کردند ، یا مقادیر انتهایی آن ها صفر است جایگزین میکنیم.

تابع check_agreement

این تابع نیز مشابه با validity است و شروط آن کاملا منطبق با سودو کد داده شده و کتاب میباشد.

```
def check_agreement():
  global false_agreement_counter
  if all(not decision for decision in decisions) or all(decisions):
    return True
  else:
    false_agreement_counter += 1
    return False
```

تابع calculate_validity_agreement_false_ratio

این تابع تنها نرخ خطاهای validity , agreement را محسابه میکند

```
def calculate_validity_agreement_false_ratio():
  validity_percentage = false_validity_counter * 1.0 / number_of_simulations
  agreement_percentage = false_agreement_counter * 1.0/ number_of_simulations
  return agreement_percentage, validity_percentage
```

تابع create_table

این تابع تنها خروجی را به فورمت خواسته شده در صورت سوال درمی‌اورد.

```
def create_table():
  # Print the table with formatting

print("┌───────┬──────────────────────┬──────────────────────┬────────────┬──────────┐"
)
  print("│ index │     value        │     decision       │ agreement  │ validity │")

print("├───────┼──────────────────────┼──────────────────────┼────────────┼──────────┤
")

  # Print each row in the table
  for i, row in enumerate(output_data):
    tindex, tvalue, tdecisions, tagreement, tvalidity = row


    if i != len(output_data) - 1:
      # Print the row with formatted and centered data
      print(f"│ {tindex:^5} │ {str(tvalue):^20} │ {str(tdecisions):^20} │ {"True" if tagreement == 1 else
"False":^10} │ {"True" if tvalidity == 1 else "False":^8} │")
    else:
      print(f"│ {"-":^5} │ {"-":^20} │ {"-":^20} │ {str(tagreement):^10} │ {str(tvalidity):^8} │")

print("└───────┴──────────────────────┴──────────────────────┴────────────┴──────────┘"
)
```

خروجی و نتیجه‌گیری

خروجی

خروجی این الگوریتم همانند تصویر زیر و منطبق با تصویری است که در صورت سوال از ما خواسته شده.

```
Enter number of rounds : 5
Enter number of nodes : 3
Enter number of simulations : 10
Enter drop messages ratio (your number / 100) : 75
```

index	value	decision	agreement	validity
0	[0, 1, 1]	[0, 0, 0]	True	True
1	[1, 1, 0]	[0, 0, 0]	True	True
2	[0, 1, 1]	[0, 0, 0]	True	True
3	[1, 1, 1]	[0, 0, 0]	True	False
4	[0, 0, 0]	[0, 0, 0]	True	True
5	[1, 1, 1]	[0, 0, 0]	True	False
6	[1, 1, 0]	[0, 0, 0]	True	True
7	[0, 0, 1]	[0, 0, 0]	True	True
8	[1, 0, 0]	[0, 0, 0]	True	True
9	[1, 0, 1]	[0, 0, 0]	True	True
-	-	-	0.0	0.2

Figure 1/استایل خروجی ـ اعداد خروجی بی معنا هستند

نتیجه گیری

در این الگوریتم با توجه به متن کتاب نرخ خطای validity , agreement با باند محدود داریم که مرتبط است با r. با استفاده از روابط ریاضی ثابت میشود که مقدار خطای disagreement برابر با $\frac{1}{(r+1)}$ است و ما در این شبیه سازی در تلاشیم تا با تولید تعداد زیادی از شبیه سازی ها و آزمایش های متفاوت به این مقدار برسیم.

آزمایش اول – چک کردن agreement و سناریو نقض شدن آن

با توجه به شروطی که agreement دارد تنها کافیست حالتی را مثال بزنیم که در آن تمام نود ها تصمیم یکسانی نگیرند. برای این کار کافیست تمام نود ها با مقدار اولیه یک شروع کنند اما شروط لازم برای تصمیم گیری بر روی ۱ پس از اخرین راند ارضا نشود تا تصمیم نهایی یک نود برابر صفر شود. برای این کار در زمان شبیه سازی مقادیر زیر را وارد میکنیم تا

خروجی مد نظر را بگیریم.

```
Enter number of rounds : 7
Enter number of nodes : 3
Enter number of simulations : 10
Enter success messages ratio (your number / 100) : 20
Enter the ratio of initial 1 (your number / 100) : 100
```

index	value	decision	agreement	validity
1	[1, 1, 1]	[0, 0, 1]	False	True
2	[1, 1, 1]	[0, 0, 0]	True	True
3	[1, 1, 1]	[0, 0, 0]	True	True
4	[1, 1, 1]	[1, 1, 0]	False	True
5	[1, 1, 1]	[0, 0, 0]	True	True
6	[1, 1, 1]	[1, 1, 1]	True	True
7	[1, 1, 1]	[1, 1, 1]	True	True
8	[1, 1, 1]	[1, 1, 1]	True	True
9	[1, 1, 1]	[0, 1, 0]	False	True
10	[1, 1, 1]	[0, 0, 0]	True	True
-	-	-	0.30000	0.00000

disagreement 2 Figure

همان طور که در تصویر پیداست در این حالت با خطای لینک 80 درصدی و مقدار اولیه 1 برای تمام نود ها ، تصمیم برخی نود ها دارای خطا شده است.

آزمایش دوم – چک کردن validity و سناریو نقض آن

این سناریو نیاز به چک کردن ندارد چون هیچ گاه ممکن نیست با این تعریف validity نقض بشود زیرا:

برای نقض آن باید نود ها با مقادیر صفر شروع کنند و با مقادیر غیر صفر کار خود را پایان دهند. این حالت ممکن نیست چون اگر نود ها از تصمیم یکدیگر مطلع نباشند آنگاه فرض را بر صفر بودن قرار میدهند و اگر از تصمیم یکدیگر مطلع شوند باز هم چون تصمیم ها صفر است حاصل فرقی نخواهد کرد.

برای نقض شرط دوم در صورتی که فرض ما کامل بودن گراف اتصال نود ها به یکدیگر است در صورتی که پیامی حذف نشود، در تنها یک راند تمام نود ها از تصمیم 1 یکدیگر مطلع خواهند شد و مقدار کلید هم بین آن ها توزیع میشود و تنها شرط کوچک بودن لول از مقدار کلید باقی میماند. چون ما r راند داریم و هیچ پیامی در این r راند نباید گم شود درنتیجه پس از

اتمام این مقدار راند حتما مقدار لول نیز بزرگتر از کلید خواهد بود زیرا کلید در بازه‌ی [1,r] قرار دارد و ماکزیمم مقدار آن نیز r است که قابل دسترسی خواهد بود

پس validity هرگز نقض نمیشود.

آزمایش سوم – چک کردن agreement برای r های زیاد

با همان سناریو آزمایش اول که منجر به نقض اگریمنت میشد حال تعداد راند ها را بالا میبریم و بررسی میکنیم که خطا به چه شکل خواهد بود.

```
Enter number of rounds : 8
Enter number of nodes : 3
Enter number of simulations : 10
Enter success messages ratio (your number / 100) : 20
Enter the ratio of initial 1 (your number / 100) : 100
```

index	value	decision	agreement	validity
1	[1, 1, 1]	[1, 1, 1]	True	True
2	[1, 1, 1]	[0, 0, 0]	True	True
3	[1, 1, 1]	[1, 1, 1]	True	True
4	[1, 1, 1]	[1, 1, 1]	True	True
5	[1, 1, 1]	[0, 0, 0]	True	True
6	[1, 1, 1]	[0, 0, 0]	True	True
7	[1, 1, 1]	[1, 0, 0]	False	True
8	[1, 1, 1]	[1, 1, 1]	True	True
9	[1, 1, 1]	[0, 0, 0]	True	True
10	[1, 1, 1]	[0, 0, 0]	True	True
-	-	-	0.10000	0.00000

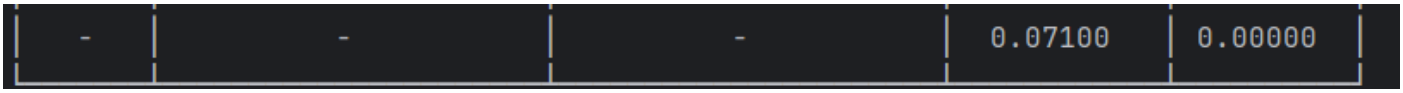
agreement and round relations3 Figure

همان طور که در تصویر پیداست تنها با یک واحد افزایش تعداد راند ها مقدار خطا در agreement یک سوم شده است.

حال همین آزمایش را با تعداد شبیه سازی های بیشتر در هر دو حالت 7 راند و 14 راند تکرار میکنیم.

-	-	-	0.13400	0.00000

7 round 1000 simulation 4 Figure



14 round 1000 simulation5 Figure

با افزایش 2 برابری تعداد راند ها مقدار disagreement تقریبا نصف شده است.

با توجه به اثبات ریاضی که داشتیم و رابطه معکوس مقدار r با مقدار خطا در agreement میتوان فهمید آزمایش ما مطابق با رابطه از قبل اثبات شده پیش میرود.