«به نام خدا»

گزارش کار تمرین کامپیوتری 2

محمد جواد بشارتی

شماره دانشجویی : 810199386

قسمت اول : ژنتیک

تعریف ژن : هر عملگر یا عملوند یک ژن است و یک مجموعه به طول معادله مورد نظر، یک کروموزوم است.

تولید هر کروموزوم به این شکل است برای خانه های شماره زوج آن (با فرض شماره گذاری خانه ها با شروع از ۰) از یک عملوند و برای خانه های شماره فرد آن از یک عملگر استفاده میکنیم و طول کروموزوم چون قرار است جوابی برای مسأله باشد باید هم اندازه با طول معادله باشد. همچنین بعد از ساخته شدن هر کروموزوم آن را به جمعیت اضافه میکنیم.

در ادامه کد های مربوطه آورده شده است :

```
for i in range(populationSize) :
    operands = random.sample(self.operands, operandsCount)
    operators = random.sample(self.operators, operatorsCount)
    chromosome = [None for i in range(self.equationLength)]
    for i in range(operandsCount) :
        chromosome[2 * i] = str(operands[i])
        if i < operandsCount - 1 :
            chromosome[2 * i + 1] = operators[i]
    population.append(chromosome)
```

تعریف تابع معیار سازگاری : معیار سازگاری نزدیک بودن مقداری که کروموزوم نشان میدهد به جواب مسأله است که برای این کار قدر مطلق اختلاف مقدار کروموزوم با عدد سمت راست معادله را به دست می آوریم و هر چه این عدد کمتر باشد، کروموزوم بهتری داریم. در ادامه کد های مربوطه آورده شده است :

```
def calcFitness(self, chromosome):
    try :
        return abs(self.goalNumber - eval(chromosome))
    except SyntaxError :
        return 10 ** 12
```

برای انجام دادن crossover یک عدد رندم در بازه طول کروموزم تولید میکنیم و بر این اساس جای ژن های دو کروموزوم را عوض میکنیم. برای mutation هم دو عدد رندم زوج برای جابجایی دو ژنی که عملوند داشته باشند و دو عدد رندم فرد برای جابجایی دو ژنی که عملگر داشته باشند تولید میکنیم و جابجایی ها را انجام میدهیم. در ادامه کد های مربوطه آورده شده است :

```python
def createCrossoverPool(self, matingPool):
    crossoverPool = []
    for i in range(0, len(matingPool), 2):
        if i == len(matingPool) - 1 :
            crossoverPool.append(matingPool[i])
            break
        if random.random() > crossoverProbability:
            crossoverPool.append(matingPool[i])
            crossoverPool.append(matingPool[i + 1])
        else:
            tmp_rand = random.randint(0, self.equationLength - 1)
            crossoverPool.append(matingPool[i][:tmp_rand] + matingPool[i + 1][tmp_rand:])
            crossoverPool.append(matingPool[i + 1][:tmp_rand] + matingPool[i][tmp_rand:])
    return crossoverPool
```

```python
def mutate(self, chromosome):
    a, b = random.randrange(0, self.equationLength, 2), random.randrange(0, self.equationLength, 2)
    c, d = random.randrange(1, self.equationLength - 1, 2), random.randrange(1, self.equationLength - 1, 2)
    chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
    chromosome[c], chromosome[d] = chromosome[d], chromosome[c]
    return chromosome
```

پاسخ به سؤالات :

۱ – اگر جمعیت اولیه بسیار کوچک باشد، احتمال رسیدن به جواب بهینه کم می‌شود و یا ممکن است رسیدن به جواب زمان زیادی طول بکشد و اگر جمعیت اولیه بسیار بزرگ باشد، سرعت محاسبه هر دوره کاهش خواهد یافت و ممکن است نتوانیم دوره های گوناگون و متنوعی را برای رسیدن به جواب آزمایش کنیم ولی ممکن است زمان رسیدن به جواب کاهش یابد.

۲ – سرعت کاهش می‌یابد ولی دقت کار افزایش خواهد یافت.

۳ – در mutation برخی ژن‌ها دچار جهش می‌شوند ولی در crossover فرزندان حاصل ترکیب ژن‌های دو والد هستند و اگر فقط crossover یا فقط mutation داشته باشیم ممکن است جمعیت به سمت تولید یک سری ژن خاص برود و از جواب بهینه دور شویم.

۴ – تعریف مناسب کروموزوم ها یا بهره بردن از یک heuristic مناسب‌تر می‌تواند سرعت الگوریتم را افزایش دهد.

۵ – دلیل این اتفاق می‌تواند این باشد که گوناگونی کروموزوم ها کاهش یابد و جمعیت به سمت تولید برخی ژن‌های خاص برود و برای رفع این مشکل :

- ۱۵٪ برتر کروموزوم ها را مستقیم به نسل بعد انتقال می‌دهیم.
- heuristic میزان نزدیکی یک کروموزوم به جواب اصلی است.
- به طور تصادفی crossover انجام می‌دهیم.

۶ – در هر مرحله نزدیک‌ترین کروموزوم‌ها به جواب اصلی را ذخیره کنیم و اگر جوابی پیدا نشد، نزدیک‌ترین کروموزوم به جواب اصلی را به عنوان جواب اعلام کنیم.

قسمت دوم : بازی

پیاده سازی الگوریتم minimax :

```python
def minimax(self, depth, player_turn, curr_choice, red_choices, blue_choices, available_choices, best_choices, alpha, beta):
    result = self.gameover(red_choices, blue_choices)
    if result == 'red' :
        return math.inf
    elif result == 'blue' :
        return -math.inf

    if depth <= 0 :
        return self._evaluate(available_choices, player_turn, red_choices, blue_choices)

    if player_turn == 'red' :
        best_choice, max_value = curr_choice, -math.inf
        for i in range(len(available_choices)) :
            tmp_available_choices = deepcopy(available_choices)
            tmp_red_choices = deepcopy(red_choices)
            curr_choice = tmp_available_choices.pop(i)
            tmp_red_choices.append(curr_choice)
            tmp_max = self.minimax(depth - 1, 'blue', curr_choice, tmp_red_choices,
            blue_choices, tmp_available_choices, best_choices, alpha, beta)
            if tmp_max > max_value :
                best_choice, max_value = curr_choice, tmp_max
                alpha = max(alpha, max_value)
                if self.prune and alpha >= beta : break
        best_choices.clear()
        best_choices.append(best_choice)
        return max_value
    elif player_turn == 'blue' :
        min_value = math.inf
        for i in range(len(available_choices)) :
            tmp_available_choices = deepcopy(available_choices)
            tmp_blue_choices = deepcopy(blue_choices)
            curr_choice = tmp_available_choices.pop(i)
            tmp_blue_choices.append(curr_choice)
            tmp_min = self.minimax(depth - 1, 'red', curr_choice, red_choices,
            tmp_blue_choices, tmp_available_choices, best_choices, alpha, beta)
            if tmp_min < min_value :
                min_value = tmp_min
                beta = min(beta, min_value)
                if self.prune and beta <= alpha :
                    break
        return min_value
```

بررسی نتایج :

عمق	هرس $\alpha\beta$	نتایج به ازای ۳ بار ران کردن کد برای ۱۰۰ بار اجرای بازی
۱	×	{'red': 96, 'blue': 4} execution time = 0.4611623287200928 seconds {'red': 99, 'blue': 1} execution time = 0.4539062976837158 seconds {'red': 97, 'blue': 3} execution time = 0.45322656631469727 seconds
۳	×	{'red': 99, 'blue': 1} execution time = 33.918630838394165 seconds {'red': 98, 'blue': 2} execution time = 34.68476843833923 seconds {'red': 99, 'blue': 1} execution time = 34.440707206726074 seconds
۵	×	{'red': 100, 'blue': 0} execution time = 2806.900805950165 seconds {'red': 96, 'blue': 4} execution time = 3120.343232154846 seconds {'red': 98, 'blue': 2} execution time = 2940.864624977112 seconds
۱	✓	{'red': 100, 'blue': 0} execution time = 0.33825230598449707 seconds {'red': 95, 'blue': 5} execution time = 0.357513427734375 seconds {'red': 99, 'blue': 1} execution time = 0.3597397804260254 seconds
۳	✓	{'red': 98, 'blue': 2} execution time = 6.3187878131866455 seconds {'red': 100, 'blue': 0} execution time = 6.101994276046753 seconds {'red': 99, 'blue': 1} execution time = 6.223140239715576 seconds
۵	✓	{'red': 95, 'blue': 5} execution time = 86.0619306564331 seconds {'red': 98, 'blue': 2} execution time = 86.64234018325806 seconds {'red': 97, 'blue': 3} execution time = 87.25236511230469 seconds
۷	✓	{'red': 95, 'blue': 5} execution time = 794.2175550460815 seconds {'red': 98, 'blue': 2} execution time = 744.5328114032745 seconds {'red': 95, 'blue': 5} execution time = 787.7286927700043 seconds

پاسخ به سؤالات :

۱ – هر چقدر تخمینی که یک heuristic میزند به ارزش واقعی آن استیت نزدیک‌تر باشد، heuristic بهتری خواهد بود. در واقع هر چقدر یک heuristic نتایج را دقیق‌تر پیش بینی کند، بهتر است.

تعریف heuristic : اگر به استیتی رسیده باشیم که برگ باشد (ببریم یا ببازیم) امتیاز آن را در صورت برد، مثبت بی‌نهایت و در صورت باخت، منفی بی‌نهایت در نظر می‌گیریم. برای استیت‌های غیر برگ (زمانی که به آخرین عمق رسیده‌ایم ولی بازی تمام نشده است)، تمام حالات ممکن بعدی را بررسی میکنیم. به ازای هر حالت، اگر پیروز شویم، به تخمین‌مان از ارزش آن استیت، یک مقدار مثبت دلخواه اضافه می‌کنیم (در اینجا ۵ واحد اضافه می‌کنیم) و اگر ببازیم، همان مقدار مثبت دلخواه را از ارزش آن استیت کم می‌کنیم (در اینجا ۵ واحد کم می‌کنیم). در ادامه کد مربوطه آورده شده است :

```
def _evaluate(self, available_choices, player_turn, red_choices, blue_choices):
    score = 0
    if player_turn == 'red' :
        for choice in available_choices :
            tmp_choices = deepcopy(blue_choices)
            tmp_choices.append(choice)
            result = self.gameover(red_choices, tmp_choices)
            if  result == 'red' : score += 5
            elif result == 'blue' : score -= 5
    if player_turn == 'blue' :
        for choice in available_choices :
            tmp_choices = deepcopy(red_choices)
            tmp_choices.append(choice)
            result = self.gameover(tmp_choices, blue_choices)
            if  result == 'red' : score += 5
            elif result == 'blue' : score -= 5
    return score
```

۲ – هر چقدر عمق جست و جو بیشتر شود، به جواب دقیق‌تری میرسیم و شانس پیروزی بیشتر می‌شود که البته زمان بیشتری هم طول خواهد کشید.

۳ – ترتیب اضافه شدن فرزندان هر گره، اگر از هرس کردن استفاده نکنیم، تأثیری در زمان انجام و پاسخ نهایی الگوریتم ندارد ولی اگر از هرس کردن استفاده کنیم، در یک ترتیب ممکن است همان شاخه‌های با عمق کم درخت هرس شوند و مقدار زیادی در زمان اجرای الگوریتم صرفه‌جویی شود و در ترتیب دیگری ممکن است چیزی هرس نشود. پس ترتیب اضافه کردن فرزندان به هر گره در زمان اجرای الگوریتم تأثیرگذار است ولی بر پاسخ نهایی تأثیری ندارد.