الگوریتم ژنتیک و مسئله هشت کوئینز

مقدمات

الگوریتم ژنتیک زیر مجموعه‌ای از الگوریتم‌های تکاملی است. این الگوریتم از سیستم بیولوژی بدن الهام گرفته شده است. در این الگوریتم نیاز است که بتوانیم هر پاسخ ممکن به مسئله را در یک ساختار مناسب نمایش دهیم (به این نمایش chromosome representation می‌گویند).

در ادامه، یک تابع نیاز است تا هر نمونه جواب (که در ساختار مناسب نمایش داده شده است) را به مقدار عددی نگاشت کند. این تابع باید میزان مناسب بودن یک پاسخ برای مسئله را نشان دهد. این تابع را اصطلاحا fitness function می‌نامند.

پس از مشخص شدن ساختار نمایش یک پاسخ در مساله و تابع ارزیابی آن، یک نسل اولیه از پاسخ‌ها تولید می‌شود و با اعمال فرآیندها ژنتیک از نسل اولیه نسل‌های بعدی به وجود می‌آید و این روند تکرار می‌گردد تا پاسخ مناسب پیدا گردد.

این الگوریتم برای یافتن پاسخ مسائلی مناسب است که فضای جستوجوی آن‌ها بسیار گسترده است و راه‌های عددی شناخته شده نمی‌توانند ما را در رسیدن به جواب کمک کند. همچنین در شرایطی که شناخت ریاضی از مسئله نداریم ولی می‌توانیم میزان مناسب بودن یک پاسخ را اندازگیری کنیم، استفاده از این روش‌ها می‌تواند در پیدا کردن پاسخ مسئله کمک کند.

پاسخ‌های بدست آمده از الگوریتم ژنتیک بهترین جواب ممکن نیست. در واقع هیچ تضمینی برای یافتن پاسخ بهینه با استفاده از این الگوریتم وجود ندارد. ولی با استفاده از این الگوریتم می‌توان به جواب‌هایی رسید که تا حد کافی مناسب باشند. در نتیجه این مطلب، استفاده از این روش در مسائلی که نیاز به پیدا کردن پاسخ بهینه مطلق است مناسب نیست ولی برای پیدا کردن یک پاسخ نزدیک به جواب بهینه می‌تواند استفاده گردد.

مراحل این الگوریتم

۱. تولید نسل اولیه Initialization
۲. انتخاب Selection
۳. عملگرهای ژنتیکی Genetic Operations

تولید نسل اولیه

در این فاز تعدادی پاسخ برای مسئله به صورت تصادفی ایجاد می‌شود (معمولا بین ۵۰ تا ۱۰۰ پاسخ).

انتخاب

از نسل قبلی تعدادی از پاسخ‌ها انتخاب می‌شوند تا مولد پاسخ‌های جدید باشند. برای این انتخاب شیوه‌های متفاوتی پیشنهاد شده است. این انتخاب می‌تواند به صورت کاملا تصادفی باشد. راه دیگر مورد استفاده می‌تواند این باشد که تعدادی از بهترین جواب‌ها را حتما در انتخاب خود داشته باشیم و دیگر پاسخ‌ها به صورت تصادفی انتخاب شوند. یک پیشنهاد دیگر آن است که به صورت تصادفی از میان جمعیت قبلی انتخاب صورت گیرد با این تفاوت که احتمال انتخاب آن پاسخ‌هایی که برای مسئله مناسب‌تر ارزیابی شده‌اند بیشتر از دیگر پاسخ‌ها باشد (به این روش چرخ رولت هم می‌گویند).

عملگرهای ژنتیکی

پس از انتخاب والدهای نسل بعد با استفاده از عملگرهای ژنتیک نسل بعدی را ایجاد می‌کنیم. یک نکته قابل مطرح آن است که تعداد پاسخ‌های هر نسل را برابر در نظر می‌گیرند و اندازه نسل را معمولا تغییر نمی‌دهند.

دو عملگر معروف برای ایجاد نسل جدید عبارت‌اند از:

۱. برش Crossover
۲. جهش Mutation

- **عمل برش** اینگونه است که از ترکیب اطلاعات دو والد یک پاسخ جدید به وجود می‌آید.

اگر یک نمایش به صورت رشته بیتی از پاسخ را درنظر بگیریم، شکل زیر می‌تواند این فرآیند را نمایش دهد.

نحوه تعریف این عملگر بسته به تعریف مسئله و چگونگی نمایش پاسخ می‌تواند به صورت‌های متفاوت تعریف گردد. برای مثال در شکل بالا، برای برش یک نقطه انتخاب شده است. این تعداد می‌تواند بیشتر از یک باشد و یا اینکه برای هر بیت به صورت جداگانه تصمیم گیری شود.

- **عمل جهش** اینگونه است که مقادیر پاسخ به صورت تصادفی با یک احتمال (معمولا این احتمال کم است.) تغییر می‌کند و مقادیر دیگری اختیار می‌کنند. برای مثال در یک نمایش دنباله بیتی می‌توان عمل جهش را به تغییر تصادفی هر کدام از بیت‌ها از یک به صفر و یا برعکس تعریف کرد.

شرایط پایان

پس از انجام عملیات بالا یک نسل جدید از پاسخ‌ها بدست می‌آید. سوال این است که در چه وقت ادامه فرآیند باید متوقف شود. برای شرط پایان پاسخ دقیقی مطرح نیست. یک پیشنهاد آن است که تعداد نسل ثابتی را بررسی گردد. برای مثال فرآیند بالا را برای ۱۰۰۰ نسل تکرار شود و بهترین پاسخ از میان آن‌ها انتخاب گردد. پیشنهاد دیگر می‌تواند آن باشد که تا رسیدن به مقدار ارزیابی fitness مورد نظر این فرآیند را تکرار شود.

پیاده سازی مسئله هشت کوئین با استفاده از الگوریتم ژنتیک

تعریف مسئله

در یک صفحه شطرنج تعداد هشت کوئین را به نحوی قرار دهید تا یک دیگر را تهدید نکنند.

این مسئله در دسته مسائل CSP (Constraint Satisfaction Problems) قرار می‌گیرد و با روش‌های متفاوتی می‌توان آن را حل کرد. در ادامه برای نمایش چگونگی استفاده از الگوریتم ژنتیک به حل این مسئله می‌پردازیم.

تعریف ساختار مناسب برای پاسخ این مسئله

راه‌های متفاوتی برای نمایش پاسخ‌های این مسئله وجود دارد. در این نوشتار مسئله با استفاده از یک بردار به طول ۸ نمایش داده شده است. مقدار هر کدام از عناصر این بردار می‌تواند بین ۰ تا ۷ باشد. تفسیر این ساختار به این نحوه است که هر درایه از بردار، محل قرار گیری کوئین در یک ستون را نمایش می‌دهد. برای مثال درایه سوم، محل قرار گیری کوئین در ستون سوم را مشخص می‌کند. این شیوه نمایش به صورت خودکار امکان بررسی جواب‌هایی که دو کوئین در یک ستون قرار می‌گیرد را حذف می‌کند.

تعریف تابع ارزیابی

برای ارزیابی هر جواب در ابتدا، تعداد تهدید دو به دو هر کوئین مشخص می‌شود. این تعداد حد اکثر ۲۸ مورد است چرا که $C(8, 2) = 28$ انتخاب دو از هشت برابر بیست هشت است. سپس این تعداد را بر مقدار حداکثر که همان ۲۸ است تقسیم می‌گردد. هر چه این مقدار بیشتر باشد پاسخ مناسب‌تری برای مسئله است.

```
fitness_function(Entity &e) {
   int threats = 0;
   // total possible threats = C(2, 8) = 28
   // this value is for 8 queens
   const int total_posible_threats = 28;
   for (int col = 0; col < Entity::COUNT_COLUMN; col++) {
      int col_val = e.get_column(col);
      for (int ptr = col + 1; ptr < Entity::COUNT_COLUMN; ptr++) {
         int ptr_val = e.get_column(ptr);
         if (col_val == ptr_val) {
            // in same row
            threats++;
            } else if (ptr - col == ptr_val - col_val) {
            // diagonal `/`
```

```
            threats++;
            } else if (ptr - col == col_val - ptr_val) {
            // diagonal `\`
            threats++;
            }
        }
    }
  return 1 - (threats / (float)total_posible_threats);
  }
```

تولید نسل اول

نسل اول به صورت کاملا تصادفی ایجاد می‌شود. برای ایجاد هر پاسخ یک عدد تصادفی بین ۰ تا ۷ در هر درایه بردار قرار می‌گیرد.

```
default_random_engine Entity::generator;
uniform_int_distribution<int> Entity::distribution(0, COUNT_ROW-1);
Entity* Entity::generate_random_entity() {
Entity *entity = new Entity();
for (int col = 0; col < COUNT_COLUMN; col++) {
int row =  distribution(generator);
entity->set_column_row(col, row);
}
return entity;
}

Entity* generate_random_population(int size) {
Entity* population =new Entity[size];
Entity* e;
for (int i = 0; i < size; i++) {
e = Entity::generate_random_entity();
population[i] = *e;
}
return population;
}
```

انتخاب

برای انتخاب از میان یک نسل از روش چرخ رولت استفاده شده است. در این روش احتمال انتخاب هر پاسخ متناسب با مقدار ارزیابی آن پاسخ است.

```
Entity* select_from_population(Entity* population, const int size) {
   float comulative_f[size];
```

```
  float normal_f;
  float comulative = 0;
  float total_f = 0;
  // calculate sum of f value for normalization
  for (int i = 0; i < size; i++)
    total_f += fittness_function(population[i]);
  // generate the commulative value
  for (int i = 0; i < size; i++) {
    normal_f = fittness_function(population[i]) / total_f;
    comulative += normal_f;
    comulative_f[i] = comulative;
  }
  // select parents randomly with
  // respect to fittness
  Entity* selected = new Entity[size];
  double max_rnd = (double)(RAND_MAX) + 1;
  for (int i = 0; i < size; i++) {
    double rnd = rand() / max_rnd;
    int ptr = 0;
    while (ptr < size - 1) {
      if (comulative_f[ptr] >= rnd)
        break;
        ptr++;
      }
    selected[i] = population[ptr];
    }
    return selected;
}
```

اعمال عملگرهای ژنتیک

در این بخش برش بر روی پاسخ‌های انتخاب شده اعمال می‌شود. این برش به این صورت تعریف گشته است که در ابتدا پاسخ‌ها به دسته‌های دو تایی تقسیم می‌گردند و سپس یک نقطه از پاسخ‌ها به صورت تصادفی انتخاب می‌گردد و اطلاعات دو پاسخ تا قبل از آن نقطه و بعد از آن جابه‌جا می‌گردد.

```
Entity* cross_over(Entity* population, const int size) {
  if (size % 2 != 0)
    throw invalid_argument("population size is not even number!");
  Entity* result = new Entity[size / 2];
  int pop_index = 0;
  for (int i = 0; i < size; i += 2) {
    Entity* xover = new Entity();
```

```
        int xover_pnt = rand() % Entity::COUNT_COLUMN;
        for (int j = 0; j < Entity::COUNT_COLUMN; j++) {
            int val;
            if (j <= xover_pnt) {
                val = population[i].get_column(j);
            } else {
                val = population[i + 1].get_column(j);
            }
            xover->set_column_row(j, val);
        }
        result[pop_index++] = *xover;
    }
    return result;
}
```

شرط پایان

چرخه ایجاد نسل جدید آنقدر تکرار می‌گردد تا یک پاسخ با مقدار ارزیابی شده یک پیدا شود.