

## تمرین دوم هوش مصنوعی

امیرحسین رجبی (۹۸۱۳۰۱۳)

۲۴ اسفند ۱۴۰۰

### سوال اول

آ) معادل الگوریتم Random walk خواهد بود زیرا اگر  $x$  تنها حالت جمعیت باشد، دو حالت والد همان  $x$  خواهند بود و در نتیجه پس از Cross over فرزند همان  $x$  خواهد شد و با جهش در یکی از حروف  $x$  به یکی از همسایه‌های  $x$  خواهیم رفت که دقیقاً همان قدم زن تصادفی است.

ب) معادل الگوریتم Hill Climbing خواهد بود زیرا در واقع بهترین همسایه حالت فعلی را انتخاب می‌کنیم و سراغ آن می‌رویم.

ج) اگر  $k$  بزرگ شود الگوریتم به این صورت خواهد بود که بسیاری از حالات و همسایه‌های آنها مورد بررسی قرار می‌گیرند و شبیه Brute force می‌شود که دائماً تعداد زیادی از بهترین‌ها را انتخاب و وجود حالت هدف را در آن‌ها بررسی می‌کند.

د) اگر دما همواره صفر باشد هیچ‌گاه به حالتی با ارزش کمتر نخواهیم رفت و دقیقاً شبیه Hill Climbing رفتار می‌کند. (دقت کنید لازم است شطر خاتمه الگوریتم تغییر کند به این که هیچ همسایه بهتری وجود ندارد و اگر نه با شرط فعلی که صفر بودن  $T$  است الگوریتم از حلقه خارج می‌شود و جواب تصادفی نخست را برمی‌گرداند.)

ه) الگوریتم به سرعت به یک مینیوم یا ماکسیمم محلی همگرا می‌شود چرا که الگوریتم تنها در صورتی از تپه پایین خواهد آمد که  $\Delta E$  کوچک باشد (هم مرتبه  $T$  باشد) و برای کاهش ارتفاع زیاد،  $(-\Delta E)$  بزرگ) عملیات پایین آمدن Reject شده و فقط با مشاهده همسایه‌ها با ارزش بزرگتر، از تپه بالا می‌رود در نتیجه به سرعت مانند Hill Climbing رفتار خواهد کرد. اگر  $T$  ثابت باشد الگوریتم از ابتدا تنها زمانی سراغ پایین آمدن از تپه می‌رود که  $\Delta E$  بسیار کوچک باشد یعنی حول همان جوابی که هست یا باقی می‌ماند یا سراغ جواب‌های بهتر می‌رود ولی خطر کاهش ارتفاع زیاد را به جان نمی‌خرد. تقریباً شبیه Hill Climbing عمل خواهد کرد.

### سوال دوم

آ) همه جایگشت‌های  $n$  کلمه خواهد بود. اگر  $n$  کلمه متمایز باشند می‌شود  $n!$  و اگر  $i$  کلمه متمایز داشته باشیم و تکرار هر کدام  $t_1, \dots, t_i$  باشند برابر  $\frac{n!}{t_1! t_2! \dots t_n!}$  خواهد بود.

ب) یک نوع می‌توان همسایگی بین جملات تعریف کرد به این صورت که دو جمله همسایه باشند اگر جای دو کلمه آنها عوض شده باشد. مثلاً جملات «این است مجازی ترم هوش مصنوعی» و «هوش مصنوعی است این ترم مجازی» همسایه جمله داده شده خواهند بود.

ج) خیر زیرا اساساً الگوریتم Hill Climbing الگوریتمی Complete نیست و ممکن است در مینیوم و ماکسیمم‌های محلی و فلات‌ها گیر کند و عملاً هیچ‌گاه به نقطه بهین سراسری نرسد.

د) اگر دو جمله «این است مجازی ترم هوش مصنوعی» و «هوش مصنوعی است این ترم مجازی» در مرحله Selection به عنوان والد انتخاب شده باشند، نتیجه عملیات Cross over با محل شکست پس از دومین کلمه، می‌تواند فرزندان روبرو باشد: «این است این ترم مجازی» و «هوش مصنوعی است مجازی ترم هوش مصنوعی».

## گزارش پیاده سازی الگوریتم Simulated Annealing

برای هر حالت مسئله یا ترتیب دهی رئوس مانند  $v_1, v_2, \dots, v_n$ ، تابع هدف  $f$  برابر تعداد یال‌هایی مانند  $v_i \rightarrow v_j$  خواهد بود که  $i < j$ ؛ یعنی تعداد یال‌ها در جهت ترتیب نهایی. هدف بیشینه کردن تابع  $f$  است. به وضوح بیشترین مقدار  $f$  برابر تعداد یال‌های گراف خواهد بود. همچنین همسایگی یک حالت مانند  $v_1, v_2, \dots, v_n$  همه حالت‌هایی خواهند بود که با جابجایی  $v_i$  و  $v_j$  بدست آمده باشند؛ یعنی  $v_1, \dots, v_j, \dots, v_i, \dots, v_n$  که  $i < j$ . در این صورت الگوریتم SA را می‌توان برای محاسبه ترتیب توپولیژیکی گراف ورودی به کار برد. تابع `read_graph()` گراف را از فایل خوانده و تعداد رئوس، تعداد یال‌ها و یک `dict` بر می‌گرداند که هر راس مانند  $u$  را به مجموعه (`set`) رئوسی نگاشت می‌کند که یالی جهت دار از  $u$  به آنها وجود داشته باشد.

```
1 def read_graph(file_name):
2     with open(file_name) as f:
3         lines = f.readlines()
4         vertex_count = int(lines[0])
5         edge_count = 0
6         graph = {}
7         for vertex in range(1, vertex_count + 1):
8             graph[vertex] = set()
9         for line in lines[1:]:
10             x, y = map(int, line.split())
11             edge_count += 1
12             graph[x].add(y)
13     return vertex_count, edge_count, graph
```

تابع `val()` مقدار تابع هدف را محاسبه می‌کند که پیاده سازی آن به صورت زیر است:

```
1 def val(graph: dict, order: list):
2     total = 0
3     for i in range(len(order)):
4         for j in range(i + 1, len(order)):
5             if order[j] in graph[order[i]]:
6                 total += 1
7     return total
```

تابع `random_successor_and_value(graph, current_order, current_value)` با ورودی گرفتن گراف مسئله و حالت فعلی مقدار تابع هدف در حالت فعلی، یک حالت تصادفی از بین همسایه‌ها انتخاب می‌کند. در واقع به تصادف دو اندیس از ترتیب توپولیژیکی فرضی فعلی را انتخاب کرده و این دو راس را `swap` می‌کند. برای بهبود سرعت الگوریتم دوباره از تابع `val()` با پیچیدگی  $\mathcal{O}(n^2)$  برای محاسبه مقدار تابع هدف در حالت همسایه استفاده نشده است و به کمک مقدار فعلی تابع هدف با الگوریتمی از مرتبه  $\mathcal{O}(n)$  مقدار  $f$  حالت مجاور برگردانده می‌شود. در این الگوریتم تغییرات تابع  $f$  روی رئوس بین  $v_i$  و  $v_j$  در ترتیب توپولیژیکی محاسبه می‌شوند چرا که جابجایی این دو راس تاثیری در باقی رئوس و تغییر تابع  $f$  به سبب آنها ندارد.

```
1 def random_successor_and_value(graph: dict, current_order: list,
2                                current_value: int):
3     i, j = sorted(random.sample(range(len(current_order)), k=2))
4     delta = 0
5     if current_order[i] in graph[current_order[j]]:
6         delta = 1
```

```

7     if current_order[j] in graph[current_order[i]]:
8         delta = -1
9     for k in range(i + 1, j):
10        if current_order[i] in graph[current_order[k]]:
11            delta += 1
12        if current_order[j] in graph[current_order[k]]:
13            delta -= 1
14        if current_order[k] in graph[current_order[i]]:
15            delta -= 1
16        if current_order[k] in graph[current_order[j]]:
17            delta += 1
18    successor = current_order.copy()
19    successor[i], successor[j] = successor[j], successor[i]
20    return successor, current_value + delta

```

در نهایت الگوریتم SA پیاده سازی شده است. در واقع یک ترتیب تصادفی برای شروع انتخاب شده و مقدر تابع هدف به کمک تابع  $val()$  محاسبه شده است. در ادامه تابعی کمکی به نام  $decide()$  تعریف شده است که به احتمال  $p$  (ورودی آن) مقدار True و به احتمال  $1 - p$  مقدار False برمیگرداند.

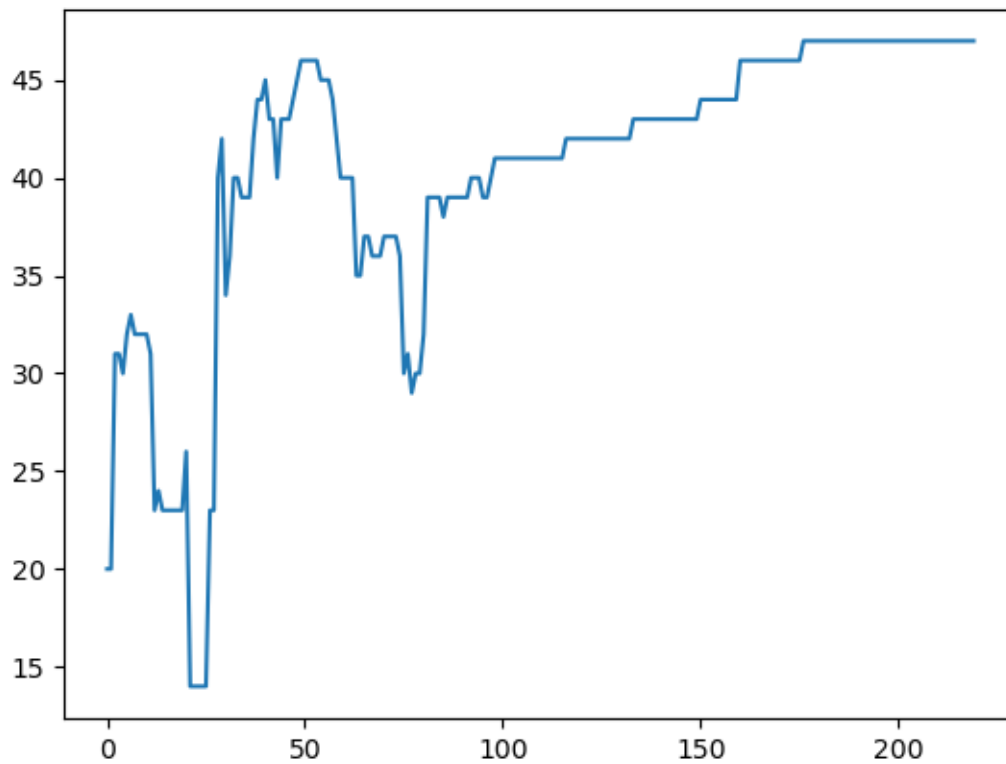
```

1 def simulated_annealing(vertex_count: int, graph: dict, temperature: int):
2     current_order = [i for i in range(1, vertex_count + 1)]
3     random.shuffle(current_order)
4     current_value = val(graph, current_order)
5     decide = lambda probability: random.random() < probability
6     print(current_order, current_value)
7     plot_data = [current_value]
8     while temperature > 1e-6:
9         successor, successor_value = random_successor_and_value(
10             graph,
11             current_order,
12             current_value
13         )
14         delta = successor_value - current_value
15         if delta >= 0 or decide(math.exp(delta / temperature)):
16             current_order = successor
17             current_value = successor_value
18             temperature *= 0.9
19             plot_data.append(current_value)
20     return current_order, current_value, plot_data

```

مابقی الگوریتم مانند الگوریتم ذکر شده در شکل پنجم از فصل چهارم کتاب پیاده سازی شده است. نمودار تابع هدف ( $current\_value$ ) براساس iterationهای الگوریتم به ازای یک بار اجرای آن روی ورودی نمونه داده شده به صورت زیر است:

	goal_function	total_edge_count
initial_order	20	54
goal_order	47	54



شکل ۱: نمودار تابع هدف (current\_value) براساس iteration