知乎 首发于 数据魔术师

【智能优化算法•爱钻研】-2迭代局部搜索算法(Iterated local search)探秘

tigerqin... 关注他

6 人赞同了该文章

> 本文发布于公众号【数据魔术师】同名文章，欢迎给我们留言或者私信一起交流
> 秦虎教授的联系方式为微信号：43340630，更多新文章请关注微信公众号：数据魔术师

内容

迭代局部搜索(Iterated local search)

字数

10分钟就能看完了

#目录#

老师，什么是迭代局部搜索?

01 局部搜索算法

1.1 什么是局部搜索算法?

局部搜索是解决最优化问题的一种启发式算法。因为对于很多复杂的问题，求解最优解的时间可能是极其长的。**因此诞生了各种启发式算法来退而求其次寻找次优解或近似最优解，局部搜索就是其中一种。它是一种近似算法（Approximate algorithms）。**

局部搜索算法是从爬山法改进而来的。简单来说，局部搜索算法是一种简单的贪心搜索算法，该算法每次从当前解的邻域解空间中选择一个最好邻居作为下次迭代的当前解，直到达到一个局部最优解(local optimal solution)。局部搜索从一个初始解出发，然后搜索解的邻域，如有更优的解则移动至该解并继续执行搜索，否则就停止算法获得局部最优解。

1.2 算法思想过程

局部搜索会先从一个初始解开始，通过**邻域动作。产生初始解的邻居解，然后根据某种策略选择邻居解。**一直重复以上过程，直到达到终止条件。

不同局部搜索算法的区别就在于：**邻域动作的定义以及选择邻居解的策略。这也是决定算法好坏的关键之处。**

1.3 什么又是邻域动作?

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

对于一个bool型问题，其当前解为：s = 1001，当将邻域动作定义为翻转其中一个bit时，得到的邻居解的集合N(s)={0001,1101,1011,1000}，其中N(s) ∈ S。同理，当将邻域动作定义为互换相邻bit时，得到的邻居解的集合N(s)={0101,1001,1010}.

02 简单局部搜索

在开始我们的迭代局部搜索之前，还是先来给大家科普几个简单局部搜索算法。他们也是基于个体的启发式算法（Single solution）。

2.1 爬山法（HILL-CLIMBING）

请阅读推文 干货 | 用模拟退火(SA, Simulated Annealing)算法解决旅行商问题

2.2 模拟退火（SIMULATED ANNEALING）

请阅读推文 干货 | 用模拟退火(SA, Simulated Annealing)算法解决旅行商问题

2.3 禁忌搜索算法(Tabu Search)

请阅读推文 干货 | 十分钟掌握禁忌搜索算法求解带时间窗的车辆路径问题(附C++代码和详细代码注释) 及 干货|十分钟快速复习禁忌搜索(c++版)

03 迭代局部搜索（Iterated Local Search, ILS）

3.1 介绍

迭代局部搜索属于探索性局部搜索方法（EXPLORATIVE LOCAL SEARCH METHODS）的一种。**它在局部搜索得到的局部最优解上，加入了扰动，然后再重新进行局部搜索。**

3.2 过程描述

注：下文的局部搜索(或者LocalSearch)指定都是内嵌的局部搜索。类似于上面介绍的几种.....

*

迭代局部搜索过程：

*** 初始状态：best_solution(最优解)、current_solution(当前解)。**

*** 从初始解(best_solution)中进行局部搜索，找到一个局部最优解s1(best_solution)。**

*** 扰动s1(best_solution)，获得新的解s2(current_solution)。**

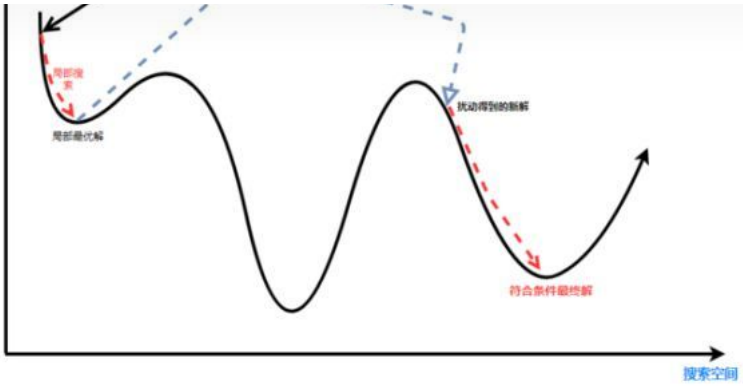
*** 从新解s2(current_solution)中进行局部搜索，再次找到一个局部最优解s3(best_solution)。**

*** 基于判断策略，对s3(current_solution)好坏进行判断。选择是否接受s3(current_solution)作为新的best_solution。**

*** 直到达到边界条件，不然跳回第二步一直循环搜索。**

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师



伪代码如下：

$s_0 \leftarrow$ GenerateInitialSolution()
$\hat{s} \leftarrow$ LocalSearch(s_0)
while termination conditions not met **do**
  $s' \leftarrow$ Perturbation($\hat{s}$, $history$)
  $\hat{s'} \leftarrow$ LocalSearch(s')
  $\hat{s} \leftarrow$ ApplyAcceptanceCriterion($\hat{s}$, $\hat{s'}$, $history$)
endwhile

04 代码时间

以下代码用于求解TSP旅行商问题。

1. //////////////////////////

2. //TSP问题 迭代局部搜索求解代码

3. //基于Berlin52例子求解

4. //作者：infinitor

5. //时间：2018-04-12

6. //////////////////////////

7.

8.

9. #include <iostream>

10. #include <cmath>

11. #include <stdlib.h>

12. #include <time.h>

13. #include <vector>

14. #include <windows.h>

15. #include <memory.h>

16. #include <string.h>

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
19. #define DEBUG

20.

21. using namespace std;

22.

23. #define CITY_SIZE 52 //城市数量

24.

25.

26. //城市坐标

27. typedef struct candidate

28. {

29. int x;

30. int y;

31. }city,
CITIES;

32.

33. //优化值

34. int **Delta;

35.

36. //解决方案

37. typedef struct Solution

38. {

39. int
permutation[CITY_SIZE]; //城市排列

40. int cost;

//该排列对应的总路线长度

41. }SOLUTION;

42. // 计算邻域操作优化值

43. int calc_delta(int i, int k, int *tmp,
CITIES * cities);

44.

45. //计算两个城市间距离

46. int distance_2city(city c1, city c2);
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
49. int cost_total(int *
cities_permutation, CITIES * cities);

50.

51. //获取随机城市排列, 用于产生初始解

52. void random_permutation(int * cities_permutation);

53.

54. //颠倒数组中下标begin到end的元素位置, 用于two_opt邻域动作

55. void swap_element(int *p, int begin, int end);

56.

57. //邻域动作 反转index_i <-> index_j 间的元素

58. void two_opt_swap(int
*cities_permutation, int *new_cities_permutation, int index_i, int index_j);

59.

60. //本地局部搜索，边界条件 max_no_improve

61. void local_search(SOLUTION & best, CITIES *
cities, int max_no_improve);

62.

63.

64.

65. //将城市序列分成4块，然后按块重新打乱顺序。

66. //用于扰动函数

67. void double_bridge_move(int
*cities_permutation, int * new_cities_permutation);

68.

69. //扰动

70. void perturbation(CITIES * cities, SOLUTION
&best_solution, SOLUTION &current_solution);

71.

72. //迭代搜索

73. void iterated_local_search(SOLUTION & best,
CITIES * cities, int max_iterations, int
max_no_improve);

74.

75. // 更新Delta
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
77.

78. //城市排列

79. int permutation[CITY_SIZE];

80. //城市坐标数组

81. CITIES
cities[CITY_SIZE];

82.

83.

84. //berlin52城市坐标，最优解7542好像

85. CITIES
berlin52[CITY_SIZE] = { { 565,575 },{ 25,185 },{ 345,750 },{ 945,685 },{ 845,655 },

86. { 880,660 },{ 25,230 },{ 525,1000 },{ 580,1175 },{ 650,1130 },{ 1605,620 },

87. { 1220,580 },{ 1465,200 },{ 1530,5 },{ 845,680 },{ 725,370 },{ 145,665 },

88. { 415,635 },{ 510,875 },{ 560,365 },{ 300,465 },{ 520,585 },{ 480,415 },

89. { 835,625 },{ 975,580 },{ 1215,245 },{ 1320,315 },{ 1250,400 },{ 660,180 },

90. { 410,250 },{ 420,555 },{ 575,665 },{ 1150,1160 },{ 700,580 },{ 685,595 },

91. { 685,610 },{ 770,610 },{ 795,645 },{ 720,635 },{ 760,650 },{ 475,960 },

92. { 95,260 },{ 875,920 },{ 700,500 },{ 555,815 },{ 830,485 },{ 1170,65 },

93. { 830,610 },{ 605,625 },{ 595,360 },{ 1340,725 },{ 1740,245 } };

94.

95. int main()

96. {

97. srand(1);

98. int
max_iterations = 600;

99. int
max_no_improve = 50;

100. //初始化指针数组

101. Delta = new int*[CITY_SIZE];

102. for (int i = 0; i <
CITY_SIZE; i ++)

103. Delta[i] = new int[CITY_SIZE];

104
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
106.

107. iterated_local_search(best_solution,
berlin52, max_iterations, max_no_improve);

108.

109. cout <<
endl<<endl<<"搜索完成！ 最优路线总长度 = " << best_solution.cost << endl;

110. cout << "最优访问城市序列如下：" <<
endl;

111. for (int i = 0; i <
CITY_SIZE;i++)

112. {

113. cout << setw(4) <<
setiosflags(ios::left) << best_solution.permutation[i];

114. }

115.

116. cout << endl << endl;

117.

118. return 0;

119. }

120.

121.

122.

123. //计算两个城市间距离

124. int distance_2city(city c1, city c2)

125. {

126. int distance
= 0;

127. distance = sqrt((double)((c1.x - c2.x)*(c1.x - c2.x) + (c1.y - c2.y)*(c1.y -
c2.y)));

128.

129. return distance;

130. }

131.

132. //根据产生的城市序列，计算旅游总距离
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

可以先访问BAC等等

知乎 首发于 数据魔术师

```
135. //p_perm 城市序列参数

136. int cost_total(int *
cities_permutation, CITIES * cities)

137. {

138. int
total_distance = 0;

139. int c1, c2;

140. //逛一圈，看看最后的总距离是多少

141. for (int i = 0; i <
CITY_SIZE; i++)

142. {

143. c1 =
cities_permutation[i];

144. if (i ==
CITY_SIZE - 1) //最后一个城市和第一个城市计算距离

145. {

146. c2 =
cities_permutation[0];

147. }

148. else

149. {

150. c2 = cities_permutation[i
+ 1];

151. }

152. total_distance +=
distance_2city(cities[c1], cities[c2]);

153. }

154.

155. return
total_distance;

156. }

157.

158. //获取随机城市排列

159. void random_permutation(int *
cities_permutation)
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
162. for (i = 0; i <
CITY_SIZE; i++)

163. {

164.
cities_permutation[i] = i; //初始化城市排列，初始按顺序排

165. }

166.

167.

168. for (i = 0; i <
CITY_SIZE; i++)

169. {

170. //城市排列顺序随机打乱

171. r = rand() %
(CITY_SIZE - i) + i;

172. temp =
cities_permutation[i];

173.
cities_permutation[i] = cities_permutation[r];

174.
cities_permutation[r] = temp;

175. }

176. }

177.

178.

179.

180.

181. //颠倒数组中下标begin到end的元素位置

182. void swap_element(int *p, int begin, int end)

183. {

184. int temp;

185. while (begin < end)

186. {

187. temp = p[begin];

188. p[begin] = p[end];
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
190. begin++;

191. end--;

192. }

193. }

194.

195.

196. //邻域动作 反转index_i <-> index_j 间的元素

197. void two_opt_swap(int
*cities_permutation, int *new_cities_permutation, int index_i, int index_j)

198. {

199. for (int i = 0; i <
CITY_SIZE; i++)

200. {

201.
new_cities_permutation[i] = cities_permutation[i];

202. }

203.

204.
swap_element(new_cities_permutation, index_i, index_j);

205. }

206.

207.

208.

209. int calc_delta(int i, int k, int *tmp,
CITIES * cities){

210. int delta = 0;

211. /*

212. 以下计算说明:

213. 对于每个方案，翻转以后没必要再次重新计算总距离

214. 只需要在翻转的头尾做个小小处理

215.

216. 比如:

217. 有城市序列 1-2-3-4-5 总距离 = d12 + d23 + d34 + d45 + d51 = A

= B
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
220. 下面的优化就是基于这种原理

221. */

222. if (i == 0)

223. {

224. if (k ==
CITY_SIZE - 1)

225. {

226. delta = 0;

227. }

228. else

229. {

230. delta
= 0

231.
- distance_2city(cities[tmp[k]], cities[tmp[k + 1]])

232.
+ distance_2city(cities[tmp[i]], cities[tmp[k + 1]])

233.
- distance_2city(cities[tmp[CITY_SIZE - 1]],
cities[tmp[i]])

234.
+ distance_2city(cities[tmp[CITY_SIZE - 1]],
cities[tmp[k]]);

235. }

236.

237. }

238. else

239. {

240. if (k ==
CITY_SIZE - 1)

241. {

242. delta
= 0

243.
- distance_2city(cities[tmp[i - 1]],
cities[tmp[i]])
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
cities[tmp[k]])

245.
- distance_2city(cities[tmp[0]], cities[tmp[k]])

246.
+ distance_2city(cities[tmp[i]], cities[tmp[0]]);

247. }

248. else

249. {

250. delta
= 0

251.
- distance_2city(cities[tmp[i - 1]],
cities[tmp[i]])

252.
+ distance_2city(cities[tmp[i - 1]],
cities[tmp[k]])

253.
- distance_2city(cities[tmp[k]], cities[tmp[k + 1]])

254.
+ distance_2city(cities[tmp[i]], cities[tmp[k + 1]]);

255. }

256. }

257.

258. return delta;

259. }

260.

261.

262. /*

263. 去重处理，对于Delta数组来说，对于城市序列1-2-3-4-5-6-7-8-9-10，如果对3-5应用了邻域操作2-opt， 事实上对于

264. 7-10之间的翻转是不需要重复计算的。 所以用Delta提前预处理一下。

265.

266. 当然由于这里的计算本身是O（1） 的，事实上并没有带来时间复杂度的减少（更新操作反而增加了复杂度）

267. 如果delta计算 是O（n）的，这种去重操作效果是明显的。

268. */
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
271. if (i
&& k != CITY_SIZE - 1){

272. i --; k ++;

273. for (int j = i; j
<= k; j ++){

274. for (int l = j + 1; l <
CITY_SIZE; l ++){

275. Delta[j][l] = calc_delta(j, l, tmp, cities);

276. }

277. }

278.

279. for (int j = 0; j <
k; j ++){

280. for (int l = i; l
<= k; l ++){

281. if (j >= l) continue;

282. Delta[j][l] = calc_delta(j, l, tmp, cities);

283. }

284. }

285. }// 如果不是边界，更新(i-1, k + 1)之间的

286. else{

287. for (i = 0; i <
CITY_SIZE - 1; i++)

288. {

289. for (k = i + 1; k <
CITY_SIZE; k++)

290. {

291. Delta[i][k] = calc_delta(i, k, tmp, cities);

292.
}

293. }

294. }// 边界要特殊更新

295.

296. }
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
299. //best_solution最优解

300. //current_solution当前解

301. void local_search(SOLUTION & best_solution,
CITIES * cities, int max_no_improve)

302. {

303. int count = 0;

304. int i, k;

305.

306. int
inital_cost = best_solution.cost; //初始花费

307.

308. int now_cost
= 0;

309.

310. SOLUTION *current_solution = new SOLUTION;
//为了防止爆栈……直接new了，你懂的

311.

312. for (i = 0; i <
CITY_SIZE - 1; i++)

313. {

314. for (k = i + 1; k <
CITY_SIZE; k++)

315. {

316. Delta[i][k] = calc_delta(i, k, best_solution.permutation,
cities);

317. }

318. }

319.

320. do

321. {

322. //枚举排列

323. for (i = 0; i <
CITY_SIZE - 1; i++)

324. {

325. for (k = i + 1; k <
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
327. //邻域动作

328.
two_opt_swap(best_solution.permutation,
current_solution->permutation, i, k);

329.
now_cost = inital_cost + Delta[i][k];

330.
current_solution->cost = now_cost;

331. if (current_solution->cost <
best_solution.cost)

332.
{

333.
count = 0; //better
cost found, so reset

334. for (int j = 0; j <
CITY_SIZE; j++)

335.
{

336.
best_solution.permutation[j] =
current_solution->permutation[j];

337.
}

338.
best_solution.cost = current_solution->cost;

339.
inital_cost = best_solution.cost;

340. Update(i, k, best_solution.permutation, cities);

341.
}

342.

343. }

344. }

345.

346. count++;

347.

348. } while (count
<= max_no_improve);
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
351.

352. //将城市序列分成4块，然后按块重新打乱顺序。

353. //用于扰动函数

354. void double_bridge_move(int
*cities_permutation, int * new_cities_permutation)

355. {

356. int
temp_perm[CITY_SIZE];

357.

358. int pos1 = 1 + rand()
% (CITY_SIZE / 4);

359. int pos2 =
pos1 + 1 + rand() % (CITY_SIZE / 4);

360. int pos3 =
pos2 + 1 + rand() % (CITY_SIZE / 4);

361.

362. int i;

363. vector<int> v;

364. //第一块

365. for (i = 0; i <
pos1; i++)

366. {

367.
v.push_back(cities_permutation[i]);

368. }

369.

370. //第二块

371. for (i =
pos3; i < CITY_SIZE; i++)

372. {

373.
v.push_back(cities_permutation[i]);

374. }

375. //第三块

376. for (i =
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

```
378. v.push_back(cities_permutation[i]);

379. }

380.

381. //第四块

382. for (i =
pos1; i < pos2; i++)

383. {

384.
v.push_back(cities_permutation[i]);

385. }

386.

387.

388. for (i = 0; i < (int)v.size();
i++)

389. {

390. new_cities_permutation[i] = v[i];

391. }

392.

393.

394. }

395.

396. //扰动

397. void perturbation(CITIES * cities, SOLUTION
&best_solution, SOLUTION &current_solution)

398. {

399.
double_bridge_move(best_solution.permutation,
current_solution.permutation);

400. current_solution.cost =
cost_total(current_solution.permutation, cities);

401. }

402.

403. //迭代搜索

404. //max_iterations用于迭代搜索次数
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

知乎 首发于 数据魔术师

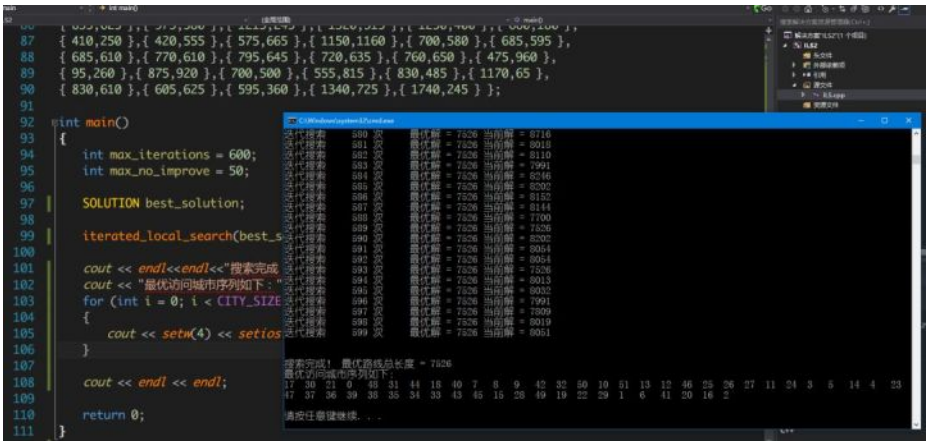
```
max_no_improve)

407. {

408. SOLUTION *current_solution = new SOLUTION;

409.

410. //获得初始随机解

411. random_permutation(best_solution.permutation);

412.

413.

414. best_solution.cost =
cost_total(best_solution.permutation, cities);

415. local_search(best_solution,
cities, max_no_improve); //初始搜索

416.

417. for (int i = 0; i <
max_iterations; i++)

418. {

419.
perturbation(cities, best_solution, *current_solution); //扰动+判断是否接受新解

420.
local_search(*current_solution, cities, max_no_improve);//继续局部搜索

421.

422. //找到更优解

423. if
(current_solution->cost < best_solution.cost)

424. {

425. for (int j = 0; j <
CITY_SIZE; j++)

426. {

427.
best_solution.permutation[j] =
current_solution->permutation[j];

428. }

429.
best_solution.cost = current_solution->cost;

430. }
```

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···

432. }

433.

434. }



写在后面

这次推文实在拖太久啦。代码从ruby改写成C++，写了好久。然后又因为优化问题改了好几次。可能是能力不够，最后差点放弃了。最后还是请教学长出马解决了最后一个优化问题。看来修仙之道还是漫漫长路啊。小编在这里希望能和大家一起努力，一起进步。

-End-

文案 /
邓发珩（大一）

排版
/ 邓发珩（大一）

代码 /
邓发珩、贺兴（大三）

指导老师 / 秦时明岳

如有疑问，欢迎咨询~

更多最新数据分析相关原创文章，请关注微信公众号：数据魔术师

知乎 首发于 数据魔术师

编辑于 2022-03-22 05:07

搜索算法 C / C++

文章被以下专栏收录

数据魔术师
有故事的地方，就有数据。

推荐阅读

搜索算法 in Python

寒冰

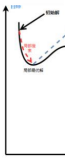
优化算法简介

论文小钢炮

凸优化: 回溯直线搜索 Backtracking line search

深度学习优化方法层出不穷，总结起来基本不外乎在 方向和步长(或者说学习率)这两方面下功夫。深度学习是非凸优化问题，本文简单介绍下凸优化中关于步长选择的一种方法：回溯直线搜索（Backt...

冷比特er 发表于机器学习:...



干货|迭 (Itera

tiger...

还没有评论

▲ 赞同 6 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···