

Отчёт по курсу «Методы решения наукоемких большеразмерных прикладных труднорешаемых задач».

Владислав Соврасов
аспирант гр. 2-о-051318

1 Сравнение различных стратегий ветвления

В ходе выполнения работы был реализован метод ветвей и границ для решения задачи о доставке заказов. Данная задача имеет $n!$ допустимых решений, лишь некоторые из которых являются оптимальными (n — количество пунктов доставки). Реализация поддерживает следующие стратегии ветвления: поиск в ширину, поиск в глубину, оптимистичная, реалистичная.

В таблице 1 приведены показатели эффективности метода на тестовых задачах с различными стратегиями ветвления. Для задачи о доставке эффективность вычисляется следующим способом: $T = 1 - \frac{N_{vis}}{n!}$, где n — размерность задачи, N_{vis} — количество обработанных методом ветвей и границ вершин.

На рассматриваемом наборе тестовых задач в среднем самым эффективным оказался метод с реалистичной стратегией ветвления. Наихудший результат показал метод, использующий стратегию поиска в ширину. В некоторых случаях решение, полученное жадным алгоритмом при построении верхней оценки оказывалось оптимальным и методу ветвей и границ требовалось обработать всего одну вершину, чтобы решить задачу. Остальные вершины при этом были сразу же отсечены.

Таблица 1: Эффективность и количество обработанных узлов при различных стратегиях ветвления

Problem	depth-first	optimistic	realistic	breadth-first
01	0.8333333333(1)	0.8333333333(1)	0.8333333333(1)	0.8333333333(1)
02	0.8333333333(1)	0.8333333333(1)	0.8333333333(1)	0.8333333333(1)
03	0.99997712743(83)	0.99998126102(68)	0.99998126102(68)	0.99997409612(94)
04	0.99998346561(60)	0.99998511905(54)	0.9999845679(56)	0.99997161596(103)
05	0.99998015873(72)	0.9999845679(56)	0.99998897707(40)	0.99997712743(83)
06	0.9999999952(628)	0.9999999996(49)	0.99999999996(48)	0.99999999557(5787)
07	0.99999999551(5869)	0.99999999759(3145)	0.99999999911(1167)	0.99999993155(89508)
08	1.0(87742)	1.0(80445)	1.0(80442)	1.0(80457)
09	1.0(1)	1.0(1)	1.0(1)	1.0(1)
10	1.0(134208)	1.0(25935)	1.0(145452)	1.0(32471)
T_{avg}	0.96666074135	0.96666176122	0.96666214717	0.96665894333

2 Исходный код

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 '''
4 Copyright (C) 2018 Sovrasov V. - All Rights Reserved
5 * You may use, distribute and modify this code under the
6 * terms of the MIT license.
7 * You should have received a copy of the MIT license with
8 * this file. If not visit https://opensource.org/licenses/MIT
9 '''
10 import math
11 import abc
12 from heapq import *
13
14 FLT_INF = float('inf')
15
16 class node:
17     def __init__(self, v=[], upper=FLT_INF, lower=FLT_INF, \
18                 partial_objective=FLT_INF, partial_time=0):
19         self.v = v
20         self.upper = upper
21         self.lower = lower
22         self.partial_objective = partial_objective
23         self.partial_time = partial_time
24
25     def __lt__(self, other):
26         return self.v < other.v
27
28 class branch_strategy:
29     def __init__(self):
30         pass
31
32     @abc.abstractmethod
33     def put(self, v):
34         pass
35
36     @abc.abstractmethod
37     def get(self):
38         pass
39
40     @abc.abstractmethod
41     def empty(self):
42         pass
43
44 class breadth_first(branch_strategy):
45     def __init__(self):
46         self.storage = []
47
```

```

48     def put(self, v):
49         self.storage.append(v)
50
51     def get(self):
52         return self.storage.pop(0)
53
54     def empty(self):
55         return len(self.storage) == 0
56
57 class depth_first(branch_strategy):
58     def __init__(self):
59         self.storage = []
60
61     def put(self, v):
62         self.storage.append(v)
63
64     def get(self):
65         return self.storage.pop()
66
67     def empty(self):
68         return len(self.storage) == 0
69
70 class optimistic(branch_strategy):
71     def __init__(self):
72         self.storage = []
73
74     def empty(self):
75         return len(self.storage) == 0
76
77     def get(self):
78         key, v = heappop(self.storage)
79         return v
80
81     def put(self, v):
82         heappush(self.storage, (v.lower, v))
83
84 class realistic(branch_strategy):
85     def __init__(self):
86         self.storage = []
87
88     def empty(self):
89         return len(self.storage) == 0
90
91     def get(self):
92         key, v = heappop(self.storage)
93         return v
94
95     def put(self, v):
96         heappush(self.storage, (v.upper, v))

```

```

97
98 def compute_partial_objective(v, problem):
99     assert v
100
101     objective = 0
102     time = problem.delays_matrix[0][v[0]]
103     if time > problem.limits[v[0] - 1]:
104         objective += 1
105     for i in range(1, len(v)):
106         time += problem.delays_matrix[v[i - 1]][v[i]]
107         if time > problem.limits[v[i] - 1]:
108             objective += 1
109
110     return time, objective
111
112 def compute_objective(v, problem):
113     assert len(v) == problem.n
114     return compute_partial_objective(v, problem)[1]
115
116 def update_partial_objective(vertex, problem):
117     assert vertex.v
118
119     t = len(vertex.v) - 1
120     if t == 0:
121         vertex.partial_time = problem.delays_matrix[0][vertex.v[0]]
122         vertex.partial_objective = 0
123     else:
124         vertex.partial_time += problem.delays_matrix[vertex.v[t - 1]][vertex.v[t]]
125     if vertex.partial_time > problem.limits[vertex.v[t] - 1]:
126         vertex.partial_objective += 1
127
128     return vertex
129
130 def compute_lower_bound_parallel(vertex, problem):
131     if len(vertex.v) == problem.n:
132         return vertex.partial_objective
133
134     not_visited = list(problem.feasible_coordinates - set(vertex.v))
135     time = vertex.partial_time
136     objective = vertex.partial_objective
137
138     for i in not_visited:
139         if time + problem.delays_matrix[vertex.v[-1]][i] > problem.limits[i - 1]:
140             objective += 1
141
142     return objective
143
144 def compute_upper_bound_greedy(vertex, problem):
145     if len(vertex.v) == problem.n:

```

```

146         return vertex.partial_objective
147
148     greedy_v = node(list(vertex.v), partial_objective=vertex.partial_objective, \
149                    partial_time=vertex.partial_time)
150     not_visited = list(problem.feasible_coordinates - set(greedy_v.v))
151
152     while len(greedy_v.v) < problem.n:
153         w = [FLT_INF]*len(not_visited)
154         z = greedy_v.partial_time
155         for i, beta in enumerate(not_visited):
156             t_d = problem.limits[beta - 1]
157             t = z + problem.delays_matrix[greedy_v.v[-1]][beta]
158             if t <= t_d:
159                 w[i] = t_d - t
160         best_idx = w.index(min(w))
161         greedy_v.v.append(not_visited.pop(best_idx))
162         greedy_v = update_partial_objective(greedy_v, problem)
163
164     return greedy_v.v, greedy_v.partial_objective
165
166 def branch(vertex, problem):
167     assert len(vertex.v) < problem.n
168     not_visited = list(problem.feasible_coordinates - set(vertex.v))
169     new_nodes = [node()]*len(not_visited)
170
171     for i, idx in enumerate(not_visited):
172         new_nodes[i] = node(vertex.v + [idx], vertex.upper, vertex.lower, \
173                            vertex.partial_objective, vertex.partial_time)
174
175     return new_nodes
176
177 branch_strategies = ['breadth-first', 'depth-first', 'optim', 'real']
178
179 def solve_transportation(problem, branch_strategy='breadth-first',
180                          compute_lower_bound=compute_lower_bound_parallel,
181                          compute_upper_bound=compute_upper_bound_greedy):
182     if branch_strategy == 'breadth-first':
183         nodes = breadth_first()
184     elif branch_strategy == 'depth-first':
185         nodes = depth_first()
186     elif branch_strategy == 'optim':
187         nodes = optimistic()
188     elif branch_strategy == 'real':
189         nodes = realistic()
190
191     n = problem.n
192     nodes.put(node([], n, n))
193     best_point = node([], n, n)
194     best_upper_bound = n

```

```

195     iters = 0
196
197     while not nodes.empty():
198         iters += 1
199         vertex = nodes.get()
200
201         if len(vertex.v) == n:
202             assert vertex.lower == vertex.upper
203             if vertex.lower < best_point.lower:
204                 best_point = vertex
205         else:
206             new_nodes = branch(vertex, problem)
207             for new_vertex in new_nodes:
208                 new_vertex = update_partial_objective(new_vertex, problem)
209                 new_vertex.lower = compute_lower_bound(new_vertex, problem)
210                 if len(new_vertex.v) < n:
211                     greedy_solution, new_vertex.upper = \
212                         compute_upper_bound(new_vertex, problem)
213                     assert len(greedy_solution) == n
214                     if new_vertex.upper < best_point.lower:
215                         best_point = node(greedy_solution, new_vertex.upper,
216                                           new_vertex.upper)
217                 else:
218                     new_vertex.upper = new_vertex.lower
219                     best_upper_bound = min(best_upper_bound, new_vertex.upper)
220                 if not (new_vertex.lower >= best_upper_bound):
221                     nodes.put(new_vertex)
222
223     return best_point.lower, best_point.v, iters

```