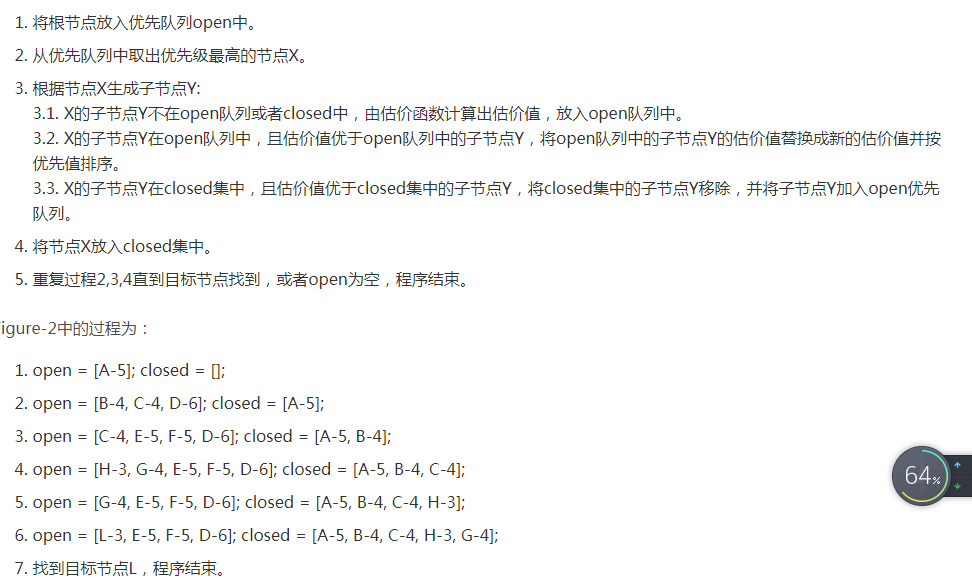
Best\_First\_Search

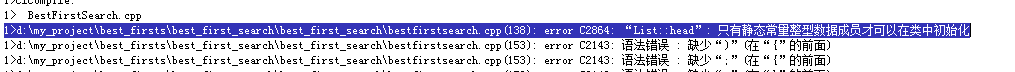
最佳优先搜索算法的实现

1. 介绍：最佳优先搜索（Best First Search），是一种启发式搜索算法（[Heuristic Algorithm](https://en.wikipedia.org/wiki/Heuristic_(computer_science))）,我们也可以将它看做广度优先搜索算法的一种改进；最佳优先搜索算法在广度优先搜索的基础上，用启发估价函数对将要被遍历到的点进行估价，然后选择代价小的进行遍历，直到找到目标节点或者遍历完所有点，算法结束。
2. 实验历程：

本次实验我参考了csdn社区的一篇文章，这篇文章介绍了最佳优先搜索算法的思路和一个实例应用，并且提供了相关的c++代码.

问题描述网页如下：<https://blog.csdn.net/qq_28781071/article/details/51548104>



原代码并不能直接运行，遇到了如下问题：

通过上网查询发现是由于编译器版本不同引起的语法变化，查询对应的用法更正即可

. 

但改正之后虽然可以成功运行，我却看不懂结果是什么意思，于是我查看了前面的讲解，理解了算法解决问题的原理：这个程序使用了链表来实现优先队列的作用.

通常采用一个open优先队列和一个closed集，open优先队列用来储存还没有遍历将要遍历的节点，而closed集用来储存已经被遍历过的节点。通过这样的数据结构，整个程序就十分清晰了

1. 实验总结：

通过本次实验，我对最佳优先搜索算法实现了基础的应用，并了解了通过链表实现队列结构的方法.

用到的代码如下：

1. #include<iostream>
2. using namespace std;
4. #define N 4
6. *//题目中所需要用到的节点*
7. class Node {
8. public:
9. Node(char \*data, int g = 0, int h = 0): data(data), g(g), h(h) {}
10. ~Node() { delete[] data; }
11. int getF() const { return g + h; }
12. int getG() const { return g; }
13. int getH() const { return h; }
14. char\* getData() const { return data; }
15. void setG(int g) { this->g = g; }
16. void setH(int h) { this->h = h; }
17. void setData(char \*data) { this->data = data; }
18. bool operator==(const Node& node) {
19. for (int i = 0; i < N; ++i) {
20. if (this->data[i] != node.getData()[i]) {
21. return false;
22. }
23. }
24. return true;
25. }
26. bool operator!=(const Node& node) {
27. for (int i = 0; i < N; ++i) {
28. if (this->data[i] != node.getData()[i]) {
29. return true;
30. }
31. }
32. return false;
33. }
34. private:
35. int g;
36. int h;
37. char\* data;
38. };
40. *//h(n)*
41. int heuristic(Node\* current, Node\* goal) {
42. int h = 0;
43. for (int i = 0; i < N; ++i) {
44. if (current->getData()[i] != goal->getData()[i]) {
45. h++;
46. }
47. }
48. return h;
49. }
51. *//链表节点*
52. struct ListNode {
53. ListNode\* next;
54. Node\* data;
55. };
57. void freeListNode(ListNode\* node) {
58. delete node->data;
59. node->data = nullptr;
60. delete node;
61. node = nullptr;
62. }
64. *//用链表实现优先队列*
65. class List {
66. public:
67. ~List() {
68. ListNode\* p = head;
69. while (p != nullptr) {
70. p = head->next;
71. freeListNode(head);
72. head = p;
73. }
74. }
75. ListNode\* getHead() { return head; }
76. void insert(ListNode\* node) {
77. if (head == nullptr) {
78. head = node;
79. node->next = nullptr;
80. } else {
81. if (node->data->getF() < head->data->getF()) {
82. node->next = head;
83. head = node;
84. } else {
85. ListNode\* p = head;
86. ListNode\* q = p->next;
87. while (q != nullptr && node->data->getF() >= q->data->getF()) {
88. p = q;
89. q = q->next;
90. }
91. p->next = node;
92. node->next = q;
93. }
94. }
95. }
96. void remove(ListNode\* node) {
97. if (head->data == node->data) {
98. head = head->next;
99. } else {
100. ListNode\* p = head;
101. ListNode\* q = head->next;
102. while (q != nullptr && q->data != node->data) {
103. p = q;
104. q = q->next;
105. }
106. if (q != nullptr && q->next != nullptr){
107. p->next = q->next;
108. q->next = nullptr;
109. freeListNode(q);
110. } else if (q->next == nullptr) {
111. p->next;
112. freeListNode(q);
113. }
114. }
115. }
116. ListNode\* findNode(ListNode\* node) {
117. ListNode\* p = head;
118. while (p != nullptr) {
119. if (\*node->data == \*p->data) {
120. return p;
121. }
122. p = p->next;
123. }
124. return nullptr;
125. }
126. void pop() {
127. ListNode\* p = head;
128. head = head->next;
129. }
130. bool empty() {
131. if (head == nullptr) {
132. return true;
133. }
134. return false;
135. }
136. private:
137. ListNode\* head = nullptr;
138. };
140. void swapChar(char& a, char& b) {
141. char temp = a;
142. a = b;
143. b = temp;
144. }
146. int main() {
148. *//初始化*
149. List open;
150. List closed;
152. Node\* goal = new Node(new char[4]{'A', 'B', 'C', 'D'});
154. ListNode\* start\_list\_node = new ListNode();
155. start\_list\_node->data = new Node(new char[4]{'C', 'B', 'A', 'D'}, 0);
156. start\_list\_node->data->setH(heuristic(start\_list\_node->data, goal));
158. open.insert(start\_list\_node);
160. while (!open.empty()) {
161. ListNode\* top = open.getHead();
162. open.pop();
164. *//输出遍历的节点*
165. cout << "[";
166. for (int i = 0; i < N; ++i) {
167. cout << top->data->getData()[i];
168. }
169. cout << "] " << top->data->getG() << "+" << top->data->getH() << endl;
171. *//找到目标结束*
172. if (\*top->data == \*goal) {
173. break;
174. }
176. *//生成子状态*
177. for (int i = N - 1; i > 0; --i) {
178. char\* temp = new char[N];
179. for (int j = 0; j < N; ++j) {
180. temp[j] = top->data->getData()[j];
181. }
182. swap(temp[i], temp[i - 1]);
183. ListNode\* child = new ListNode();
184. child->data = new Node(temp, top->data->getG()+1);
185. child->data->setH(heuristic(child->data, goal));
186. if (open.findNode(child) == nullptr && closed.findNode(child) == nullptr) {
187. open.insert(child);
188. } else if (open.findNode(child) != nullptr) {
189. ListNode\* old = open.findNode(child);
190. if (child->data->getF() < old->data->getF()) {
191. open.remove(old);
192. open.insert(child);
193. }
194. } else if (closed.findNode(child) != nullptr) {
195. ListNode\* old = closed.findNode(child);
196. if (child->data->getF() < old->data->getF()) {
197. closed.remove(old);
198. open.insert(child);
199. }
200. }
201. }
202. closed.insert(top);
203. }
205. delete goal;
206. }