## LÝ THUYẾT C++ | GIẢI THUẬT ĐỒ THỊ CÀI ĐẶT BẰNG C++

Trong topic này mình sẽ post những những thuật toán cơ bản trong đồ thị, full-code được cài đặt bằng C++. Vì mình nhận thấy lý thuyết đồ thị khá trừu tượng và rất khó cài khi ta chưa hiểu rõ bản chất. Mình hi vọng qua topic này, các bạn có thể sử dụng những đoạn code sau để hiểu rõ hơn thuật toán. Tuy nhiên lời khuyên chân thành của mìh đối với các bạn mới học là đừng chỉ copy code thôi, mà hãy giành ra chút thời gian để nghịch với chúng. Sau khi đã hiểu rõ rồi, thì hãy tự viết lại thì sẽ tốt hơn.   
Một lưu ý nhỏ là về phần lý thuyết mình chỉ tóm tắt sơ lược và để link tới wiki cho các bạn tìm và đọc thêm vì tài liệu về đồ thị trên mạng rất nhiều và kĩ năng viết của mình cũng chẳng có hay gì cho mấy. Tuy nhiên nếu bạn nào có thắc mắc thì cứ hỏi, mình sẽ cố gắng trả lời trong sự hiểu biết của mình.  
  
*@ps: Nếu có bug trong đoạn code nào thì các bạn cứ báo cho mình biết, mình sẽ fix.*  
  
**Update 01/09/2013**  
1) Khi xưa mình chuyển code từ Pascal sang C++ nên code khá rối răm và khó sử dụng, vì vậy mình quyết định sẽ viết lại toàn bộ các giải thuật trong topic này sử dụng chuẩn C++11.  
2) Đồng thời mình sẽ thêm vào cách giải thuật khác như Prim, Kruskal (Minimum Spanning Tree) và các ứng dụng của những giải thuật này để chúng ta có thể hiểu rõ hơn bản chất của chúng.  
3) Lưu ý để biên dịch các đoạn chương trình sau thì bạn phải dùng ít nhất là gcc/g++ bản 4.5 trở lên.  
  
Sau đây là một số biến quan trọng được dùng phần lớn cho các giải thuật:  
**1. Danh sách cạnh kề:**

1. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**

Ở đây nếu chúng ta có adj[u][v] nghĩa là có một cạnh từ đỉnh u sang đỉnh v.  
  
**2. Thêm cạnh vào đồ thị:**  
- Vô hướng (undirected)

1. adj[u].push\_back(v)**;**
2. adj[v].push\_back(u)**;**

- Có hướng (directed)

1. adj[u].push\_back(v)**;**

**3. Đánh dấu đỉnh đã duyệt:**

1. **bool** seen[MAX\_VERTEX]**;**

Nếu seen[u] == true thì đỉnh u đã được duyệt qua  
  
**4. Ma trận chi phí cho các bài toán tìm đường đi ngắn nhất**

1. **const** **int** oo = std**::**numeric\_limits<**int**>**::**max()**;**
2. std**::**vector<std**::**vector<**int**>> cost**;**

Ví dụ cost[u][v] = 10, có nghĩa là chi phí cho cạnh đi từ u sang v tốn 10 unit.

**Breadth First Search - Graph Traversal**  
Mở đầu sẽ là thuật toán tìm kiếm theo chiều rộng dựa trên sự vận hành của queue (FIFO)  
**I. Lý thuyết**  
<http://en.wikipedia.org/wiki/Breadth-first_search>  
  
**II. Cài đặt bằng C++**

1. #include <iostream>
2. #include <queue>
3. #include <vector>
5. **const** **int** MAX\_VERTEX = 100**;**
6. **bool** seen[MAX\_VERTEX]**;**
7. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
9. **void** visit(**int** u) {
10. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << u << "\n"**;**
11. }
13. **void** bfs(**int** s) {
14. std**::**queue<**int**> q**;**
15. q.push(s)**;**
16. seen[s] = **true;**
17. **while** (!q.empty()) {
18. **int** u = q.front()**;**
19. visit(u)**;**
20. q.pop()**;**
21. **for** (**auto** v **:** adj[u]) {
22. **if** (!seen[v]) {
23. seen[v] = **true;**
24. q.push(v)**;**
25. }
26. }
27. }
28. }
30. */\*\**
31. *\* For undirected graph*
32. *\*/*
33. **void** add\_edge(**int** u, **int** v) {
34. adj[u].push\_back(v)**;**
35. adj[v].push\_back(u)**;**
36. }
38. **int** main() {
39. add\_edge(0, 1)**;**
40. add\_edge(0, 2)**;**
41. add\_edge(0, 3)**;**
42. add\_edge(1, 4)**;**
43. add\_edge(1, 5)**;**
44. add\_edge(2, 3)**;**
45. bfs(0)**;**
46. **return** 0**;**
47. }

**III. Ứng dụng**  
**1. Tìm đường đi ngắn nhất (số cạnh) cho từng cặp đỉnh**

1. #include <iostream>
2. #include <queue>
3. #include <vector>
4. #include <limits>
6. **const** **int** MAX\_VERTEX = 100**;**
7. **const** **int** oo = std**::**numeric\_limits<**int**>**::**max()**;**
8. **bool** seen[MAX\_VERTEX]**;**
9. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
10. std**::**vector<**int**> distance(MAX\_VERTEX, oo)**;**
12. **void** bfs(**int** s) {
13. std**::**queue<**int**> q**;**
14. q.push(s)**;**
15. seen[s] = **true;**
16. distance[s] = 0**;**
17. **while** (!q.empty()) {
18. **int** u = q.front()**;**
19. q.pop()**;**
20. **for** (**auto** v **:** adj[u]) {
21. **if** (!seen[v]) {
22. *// recompute distance to v*
23. distance[v] = distance[u] + 1**;**
24. seen[v] = **true;**
25. q.push(v)**;**
26. }
27. }
28. }
29. }
31. */\*\**
32. *\* For undirected graph*
33. *\*/*
34. **void** add\_edge(**int** u, **int** v) {
35. adj[u].push\_back(v)**;**
36. adj[v].push\_back(u)**;**
37. }
39. **void** shortest\_distance() {
40. add\_edge(0, 1)**;**
41. add\_edge(0, 2)**;**
42. add\_edge(0, 3)**;**
43. add\_edge(1, 4)**;**
44. add\_edge(4, 5)**;**
45. bfs(0)**;**
46. **for** (**int** v = 0**;** v < 6**;** ++v) {
47. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "shortest distance from 0->" << v << ": " << distance[v] << std**::**endl**;**
48. }
49. }
51. **int** main() {
52. shortest\_distance()**;**
53. **return** 0**;**
54. }

**2. Check bipartite graph (2-coloring)**

1. #include <iostream>
2. #include <queue>
3. #include <vector>
4. #include <limits>
6. **enum** **class** color {
7. NONE, BLACK, WHITE
8. }**;**
10. **const** **int** MAX\_VERTEX = 100**;**
11. **const** **int** oo = std**::**numeric\_limits<**int**>**::**max()**;**
12. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
13. std**::**vector<color> colors(MAX\_VERTEX, color**::**NONE)**;**
14. **bool** is\_bipartite = **true;**
16. color other(**const** color &c) {
17. **if** (c == color**::**NONE) {
18. **return** c**;**
19. }
20. **return** (c == color**::**BLACK) **?** color**::**WHITE **:** color**::**BLACK**;**
21. }
23. **void** bfs(**int** s) {
24. std**::**queue<**int**> q**;**
25. q.push(s)**;**
26. colors[s] = color**::**WHITE**;**
27. **while** (!q.empty()) {
28. **int** u = q.front()**;**
29. q.pop()**;**
30. **for** (**auto** v **:** adj[u]) {
31. **if** (colors[v] == color**::**NONE) {
32. colors[v] = other(colors[u])**;**
33. q.push(v)**;**
34. } **else** **if** (colors[v] == colors[u]) {
35. is\_bipartite = **false;**
36. **break;**
37. }
38. }
39. }
40. }
42. **void** add\_edge(**int** u, **int** v) {
43. adj[u].push\_back(v)**;**
44. adj[v].push\_back(u)**;**
45. }
47. **void** check\_bipartite() {
48. add\_edge(0, 3)**;**    add\_edge(0, 4)**;**    add\_edge(0, 5)**;**    add\_edge(0, 6)**;**
50. *// add this edge will break bipartite property*
51. *// add\_edge(0, 1);*
53. add\_edge(1, 3)**;**    add\_edge(1, 4)**;**    add\_edge(1, 5)**;**    add\_edge(1, 6)**;**
55. add\_edge(2, 3)**;**    add\_edge(2, 4)**;**    add\_edge(2, 5)**;**    add\_edge(2, 6)**;**
56. bfs(0)**;**
57. **if** (is\_bipartite) {
58. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "G is bipartite." << std**::**endl**;**
59. } **else** {
60. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "G is non-bipartite." << std**::**endl**;**
61. }
62. }
64. **int** main() {
65. check\_bipartite()**;**
66. **return** 0**;**
67. }

**Depth First Search - Graph Traversal**  
Thuật toán kế tiếp là thuật toán tìm kiếm theo chiều sâu (DFS), nó cũng là một thuật toán dùng để viếng thăm các đỉnh của đồ thị. Điều khác biệt rõ ràng nhất với BFS là DFS hoạt động dựa trên sự vận hành của stack (FILO) thay vì queue (FIFO). Vì bản chất của DFS là dùng lời gọi đệ qui để gọi hàm, cho nên mình sẽ implement DFS dùng đệ qui thay vì dùng stack.   
  
**I. Lý thuyết**  
<http://en.wikipedia.org/wiki/Depth-first_search>  
  
**II. Cài đặt bằng C++**

1. #include <iostream>
2. #include <vector>
4. **const** **int** MAX\_VERTEX = 100**;**
5. **bool** seen[MAX\_VERTEX]**;**
6. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
8. **void** visit(**int** u) {
9. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << u << "\n"**;**
10. }
12. **void** dfs(**int** u) {
13. visit(u)**;**
14. seen[u] = **true;**
15. **for** (**auto** v **:** adj[u]) {
16. **if** (!seen[v]) {
17. dfs(v)**;**
18. }
19. }
20. }
22. */\*\**
23. *\* For undirected graph*
24. *\*/*
25. **void** add\_edge(**int** u, **int** v) {
26. adj[u].push\_back(v)**;**
27. adj[v].push\_back(u)**;**
28. }
30. **int** main() {
31. add\_edge(0, 1)**;**
32. add\_edge(0, 2)**;**
33. add\_edge(0, 3)**;**
34. add\_edge(1, 4)**;**
35. add\_edge(1, 5)**;**
36. add\_edge(2, 3)**;**
37. dfs(0)**;**
38. **return** 0**;**
39. }

**III. Ứng dụng**  
**1. Tìm thành phần liên thông của đồ thị (connected components)**

1. #include <iostream>
2. #include <vector>
4. **const** **int** MAX\_VERTEX = 100**;**
5. **bool** seen[MAX\_VERTEX]**;**
6. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
8. **void** dfs(**int** u) {
9. seen[u] = **true;**
10. **for** (**auto** v **:** adj[u]) {
11. **if** (!seen[v]) {
12. dfs(v)**;**
13. }
14. }
15. }
17. **int** count\_connected\_components(**int** no\_vertex) {
18. **int** cnt = 0**;**
19. *// for all vertex that has not been visited*
20. **for** (**int** v = 0**;** v < no\_vertex**;** ++v) {
21. **if** (!seen[v]) {
22. *// increase the number of components*
23. cnt++**;**
24. *// visit*
25. dfs(v)**;**
26. }
27. }
28. **return** cnt**;**
29. }
31. **void** add\_edge(**int** u, **int** v) {
32. adj[u].push\_back(v)**;**
33. adj[v].push\_back(u)**;**
34. }
36. **int** main() {
37. add\_edge(0, 1)**;**
38. add\_edge(0, 2)**;**
39. add\_edge(3, 4)**;**
40. add\_edge(4, 5)**;**
41. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << count\_connected\_components(6) << std**::**endl**;**
42. **return** 0**;**
43. }

**2. Toposort**

1. #include <iostream>
2. #include <vector>
4. **const** **int** MAX\_VERTEX = 100**;**
5. **bool** seen[MAX\_VERTEX]**;**
6. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
7. std**::**vector<**int**> topo\_order**;**
9. **void** dfs(**int** u) {
10. seen[u] = **true;**
11. **for** (**auto** v **:** adj[u]) {
12. **if** (!seen[v]) {
13. dfs(v)**;**
14. }
15. }
16. *// keep track the order of visited*
17. topo\_order.push\_back(u)**;**
18. }
20. **void** add\_edge(**int** u, **int** v) {
21. adj[u].push\_back(v)**;**
22. adj[v].push\_back(u)**;**
23. }
25. **void** toposort(**int** no\_vertex) {
26. **for** (**int** v = 0**;** v < no\_vertex**;** ++v) {
27. **if** (!seen[v]) {
28. dfs(v)**;**
29. }
30. }
31. *// print out topo order*
32. **for** (**auto** v **:** topo\_order) {
33. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << v << " "**;**
34. }
35. }
37. **int** main() {
38. add\_edge(0, 1)**;**
39. add\_edge(0, 2)**;**
40. add\_edge(3, 4)**;**
41. add\_edge(4, 5)**;**
42. toposort(6)**;**
43. **return** 0**;**
44. }

**3. Cycle detection**

1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
5. **enum** **class** color {
6. WHITE, GRAY, BLACK
7. }**;**
9. **const** **int** MAX\_VERTEX = 100**;**
10. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
11. std**::**vector<color> colors(MAX\_VERTEX)**;**
12. **bool** is\_cyclic = **false;**
14. **void** dfs(**int** u) {
15. *// start exploring*
16. colors[u] = color**::**GRAY**;**
17. **for** (**auto** v **:** adj[u]) {
18. **if** (colors[v] == color**::**WHITE) {
19. dfs(v)**;**
20. } **else** {
21. *// check if back-edge exists*
22. **if** (colors[v] == color**::**BLACK && colors[u] == color**::**GRAY) {
23. is\_cyclic = **true;**
24. **return;**
25. }
26. }
27. }
28. *// finish exploring*
29. colors[u] = color**::**BLACK**;**
30. }
32. **void** add\_edge(**int** u, **int** v) {
33. adj[u].push\_back(v)**;**
34. adj[v].push\_back(u)**;**
35. }
37. **void** check\_cycle(**int** no\_vertex) {
38. *// initially, all vertex are not visited*
39. std**::**fill(colors.begin(), colors.end(), color**::**WHITE)**;**
40. is\_cyclic = **false;**
41. **for** (**int** v = 0**;** v < no\_vertex**;** ++v) {
42. **if** (colors[v] == color**::**WHITE) {
43. dfs(v)**;**
44. }
45. }
46. **if** (is\_cyclic) {
47. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "There is a cycle in this graph." << std**::**endl**;**
48. } **else** {
49. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "No cycle found." << std**::**endl**;**
50. }
51. }
53. **void** test\_cycle() {
54. add\_edge(0, 1)**;**
55. add\_edge(1, 2)**;**
56. add\_edge(2, 0)**;**
57. add\_edge(4, 5)**;**
58. check\_cycle(6)**;**
59. }
61. **void** test\_non\_cycle() {
62. add\_edge(0, 1)**;**
63. add\_edge(1, 2)**;**
64. add\_edge(4, 5)**;**
65. check\_cycle(6)**;**
66. }
68. **int** main() {
69. test\_non\_cycle()**;**
70. *// test\_cycle();*
71. **return** 0**;**
72. }

**Roy Warshall's Algorithm - Shortest Path Problem**  
Thuật toán tiếp theo có tên là Floyd Warshall dùng để tìm đường với chi phi thấp nhất từ 2 cặp đỉnh (u, v). Running time cho giải thuật này là O(n^3) khá chậm so với Dijsktra, tuy nhiên nếu chúng ta cần query nhiều lần với những cặp đỉnh khác nhau thì việc precompute trước cho toàn bộ các cặp đỉnh thì thuật toán này sẽ nhanh hơn Dijkstra rất nhiều.  
Giả sử chúng ta đồ thị với V đỉnh, E cạnh, và M query thì:  
- Floyd Warshall sẽ có running time O(V^3) + M  
- Dijsktra Heap: M \* O(E \* log(V))  
  
**I. Lý thuyết**  
[http://en.wikipedia.org/wiki/Floyd%E...hall\_algorithm](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm)  
  
**II. Cài đặt bằng C++**

1. #include <iostream>
2. #include <vector>
3. #include <limits>
5. **const** **int** oo = std**::**numeric\_limits<**int**>**::**max()**;**
6. std**::**vector<std**::**vector<**int**>> cost**;**
8. **void** floyd\_warshall(**int** no\_vertex) {
9. **for** (**int** bridge = 0**;** bridge < no\_vertex**;** ++bridge) {
10. **for** (**int** u = 0**;** u < no\_vertex**;** ++u) {
11. **for** (**int** v = 0**;** v < no\_vertex**;** ++v) {
12. cost[u][v] = std**::**min(cost[u][v], cost[u][bridge] + cost[bridge][v])**;**
13. }
14. }
15. }
16. }
18. std**::**vector<std**::**vector<**int**>> generate\_sample\_cost() {
19. **return** std**::**vector<std**::**vector<**int**>> {
20. std**::**vector<**int**> {0, 2, 1, 3},
21. std**::**vector<**int**> {1, 0, 4, 5},
22. std**::**vector<**int**> {3, 1, 0, 3},
23. std**::**vector<**int**> {1, 1, 1, 0},
24. }**;**
25. }
27. **void** query(**int** u, **int** v) {
28. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "shortest path from " << u << "->" << v << ": " << cost[u][v] << "\n"**;**
29. }
31. **int** main() {
32. cost = generate\_sample\_cost()**;**
33. floyd\_warshall(cost.size())**;**
34. query(0, 2)**;**
35. query(1, 2)**;**
36. **return** 0**;**
37. }

**Prim's Algorithm - Minimum Spanning Tree**  
Thuật toán tiếp theo là thuật toán Prim, dùng để giải bài toán tìm cây khung nhỏ nhất trong đồ thị vô hướng. Thuật toán Prim là một dạng của thuật toán "tham lam" (greedy) vì nó luôn chọn cạnh với chi phí thấp nhất nhờ vào sự trợ giúp của min-heap. Về phần lý thuyết, các bạn có thể tham khảo những link sau.  
**I. Lý thuyết**  
<http://en.wikipedia.org/wiki/Prim's_algorithm>  
<http://en.wikipedia.org/wiki/Minimum_spanning_tree>  
  
**II. Cài đặt bằng C++**

1. #include <algorithm>
2. #include <iostream>
3. #include <queue>
4. #include <vector>
5. #include <utility>
7. **struct** edge {
8. **int** v**;**
9. **int** cost**;**
11. edge(**int** v, **int** cost)
12. **:**v(v), cost(cost) {
13. }
15. *// sort by cost first then vertex*
16. **bool** **operator** >(**const** edge &e) **const** {
17. **return** (cost != e.cost **?** cost > e.cost **:** v > e.v)**;**
18. }
19. }**;**
21. *// alias for min priority queue*
22. **typedef** std**::**priority\_queue<edge, std**::**vector<edge>, std**::**greater<edge>> min\_heap**;**
23. *// infinity*
24. **const** **int** oo = std**::**numeric\_limits<**int**>**::**max()**;**
25. *// maximum # of vertex*
26. **const** **int** MAX\_VERTEX = 100**;**
27. *// mark array*
28. **bool** seen[MAX\_VERTEX]**;**
29. *// adjacency list*
30. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
31. *// cost matrix, initially all cost are infinity*
32. std**::**vector<std**::**vector<**int**>> cost(MAX\_VERTEX, std**::**vector<**int**>(MAX\_VERTEX, oo))**;**
33. *// minimum cost heap*
34. min\_heap pq**;**
36. */\*\**
37. *\* For all vertex that are adjacent to u,*
38. *\* if it's not visited, we add them to*
39. *\* our queue with its cost*
40. *\*/*
41. **void** relax(**int** u) {
42. seen[u] = **true;**
43. **for** (**auto** v **:** adj[u]) {
44. **if** (!seen[v]) {
45. pq.push(edge(v, cost[u][v]))**;**
46. }
47. }
48. }
50. */\*\**
51. *\* Prim algorithm for finding the minimum*
52. *\* spanning tree of undirected graph*
53. *\*/*
54. **int** prim(**int** s) {
55. **int** min\_cost = 0**;**
56. relax(s)**;**
57. **while** (!pq.empty()) {
58. *// get the minimum cost edge*
59. edge e = pq.top()**;**
60. pq.pop()**;**
61. *// if we have seen v*
62. *// visit v, and update new cost*
63. **if** (!seen[e.v]) {
64. min\_cost += e.cost**;**
65. relax(e.v)**;**
66. }
67. }
68. **return** min\_cost**;**
69. }
71. **void** add\_edge(**int** u, **int** v, **int** c) {
72. adj[u].push\_back(v)**;**
73. adj[v].push\_back(u)**;**
74. cost[u][v] = c**;**
75. cost[v][u] = c**;**
76. }
78. **void** test\_minimum\_spanning\_tree() {
79. add\_edge(0, 1, 3)**;**
80. add\_edge(0, 2, 3)**;**
81. add\_edge(0, 3, 1)**;**
82. add\_edge(0, 4, 2)**;**
83. add\_edge(1, 2, 2)**;**
84. add\_edge(2, 3, 11)**;**
85. add\_edge(3, 4, 9)**;**
86. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "min cost: " << prim(0) << std**::**endl**;**
87. }
89. **int** main() {
90. test\_minimum\_spanning\_tree()**;**
91. **return** 0**;**
92. }

**Kruskal's Algorithm - Minimum Spanning Tree**  
Thuật toán này cũng là một dạng thuật toán tham lam dùng để giải bài toán cây khung nhỏ nhất (giống như Prim). Điểm khác biệt giữa Kruskal và Prim là Kruskal hoạt động dựa trên cấu trúc dữ liệu có tên là "Disjoint Set".   
  
**I. Lý thuyết**  
<http://en.wikipedia.org/wiki/Kruskal's_algorithm>  
[http://en.wikipedia.org/wiki/Disjoin...data\_structure](http://en.wikipedia.org/wiki/Disjoint-set_data_structure)  
  
**II. Cài đặt bằng C++**

1. #include <algorithm>
2. #include <iostream>
3. #include <vector>
4. #include <utility>
6. **const** **int** MAX\_SIZE = 100**;**
8. **class** disjoint\_set {
9. **private:**
10. **int** components**;**
11. **int** id[MAX\_SIZE]**;**
12. **int** sizeof\_ids[MAX\_SIZE]**;**
13. **int** n**;**
15. **public:**
16. disjoint\_set(**int** n)
17. **:**components(n), n(n) {
18. **for** (**int** i = 0**;** i < components**;** ++i) {
19. id[i] = i**;**
20. sizeof\_ids[i] = 1**;**
21. }
22. }
24. **void** join(**int** p, **int** q) {
25. **int** i = find(p)**;**
26. **int** j = find(q)**;**
27. **if** (i == j) {
28. **return;**
29. }
30. **if** (sizeof\_ids[i] < sizeof\_ids[j]) {
31. id[i] = j**;**
32. sizeof\_ids[j] += sizeof\_ids[i]**;**
33. sizeof\_ids[i] = 1**;**
34. } **else** {
35. id[j] = i**;**
36. sizeof\_ids[i] += sizeof\_ids[j]**;**
37. sizeof\_ids[j] = 1**;**
38. }
39. components--**;**
40. }
42. **int** find(**int** p) {
43. **if** (p != id[p]) {
44. id[p] = find(id[p])**;**
45. }
46. **return** id[p]**;**
47. }
49. **bool** is\_connected(**int** p, **int** q) {
50. **return** find(p) == find(q)**;**
51. }
53. **int** size() **const** {
54. **return** components**;**
55. }
56. }**;**
58. **struct** edge {
59. **int** u**;**
60. **int** v**;**
61. **int** cost**;**
63. edge(**int** u, **int** v, **int** cost)
64. **:**u(u), v(v), cost(cost) {
65. }
67. *// sort by cost*
68. **bool** **operator** <(**const** edge &e) **const** {
69. **return** cost < e.cost**;**
70. }
71. }**;**
73. std**::**vector<edge> edges**;**
75. **int** kruskal(**int** no\_vertex) {
76. **int** min\_cost = 0**;**
77. std**::**sort(edges.begin(), edges.end())**;**
78. disjoint\_set ds(no\_vertex)**;**
79. **for** (**auto** e **:** edges) {
80. **if** (!ds.is\_connected(e.u, e.v)) {
81. min\_cost += e.cost**;**
82. ds.join(e.u, e.v)**;**
83. }
84. }
85. **return** min\_cost**;**
86. }
88. **void** test\_minimum\_spanning\_tree() {
89. edges.push\_back(edge(0, 1, 3))**;**
90. edges.push\_back(edge(0, 2, 3))**;**
91. edges.push\_back(edge(0, 3, 1))**;**
92. edges.push\_back(edge(0, 4, 2))**;**
93. edges.push\_back(edge(1, 2, 2))**;**
94. edges.push\_back(edge(2, 3, 11))**;**
95. edges.push\_back(edge(3, 4, 9))**;**
96. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "min cost: " << kruskal(5) << std**::**endl**;**
97. }
99. **int** main() {
100. test\_minimum\_spanning\_tree()**;**
101. **return** 0**;**
102. }

**Dijkstra's Algorithm - Shortest Path Problem**  
Tiếp theo là giải thuật Dijkstra dùng để giải bài toán tìm đường đi ngắn nhất giữa các đỉnh trên đồ thị có trọng số. Cho đồ thị G = (V, E) thì Dijkstra có running time là O(E log(V)).  
  
**I. Lý thuyết**  
<http://en.wikipedia.org/wiki/Dijkstra's_algorithm>  
  
**II. Cài đặt bằng C++**

1. #include <iostream>
2. #include <vector>
3. #include <limits>
5. #include <iostream>
6. #include <queue>
7. #include <vector>
8. #include <limits>
9. #include <queue>
11. *// priority queue*
12. **typedef** std**::**priority\_queue<std**::**pair<**int**, **int**>, std**::**vector<std**::**pair<**int**, **int**> >, std**::**greater<std**::**pair<**int**, **int**>> > min\_heap**;**
13. *// cost and vertex*
14. **typedef** std**::**pair<**int**, **int**> vertex**;**
16. **const** **int** oo = std**::**numeric\_limits<**int**>**::**max()**;**
17. **const** **int** MAX\_VERTEX = 100**;**
18. std**::**vector<std**::**vector<**int**>> adj(MAX\_VERTEX)**;**
19. std**::**vector<std**::**vector<**int**>> cost(MAX\_VERTEX, std**::**vector<**int**>(MAX\_VERTEX, oo))**;**
20. std**::**vector<**int**> dist(MAX\_VERTEX, oo)**;**
22. **void** dijkstra(**int** s) {
23. min\_heap pq**;**
24. dist[s] = 0**;**
25. pq.push(vertex(0, s))**;**
26. **while** (!pq.empty()) {
27. **int** u = pq.top().second**;**
28. **int** cost\_to\_u = pq.top().first**;**
29. pq.pop()**;**
30. **if** (dist[u] == cost\_to\_u) {
31. **for** (**auto** v **:** adj[u]) {
32. **if** (dist[v] > dist[u] + cost[u][v]) {
33. dist[v] = dist[u] + cost[u][v]**;**
34. pq.push(vertex(dist[v], v))**;**
35. }
36. }
37. }
38. }
39. }
41. **void** add\_edge(**int** u, **int** v, **int** c) {
42. adj[u].push\_back(v)**;**
43. cost[u][v] = c**;**
44. }
46. **int** main() {
47. add\_edge(0, 1, 13)**;**
48. add\_edge(1, 2, 8)**;**
49. add\_edge(2, 4, 1)**;**
50. add\_edge(0, 2, 18)**;**
51. add\_edge(0, 3, 20)**;**
52. add\_edge(0, 4, 23)**;**
53. dijkstra(0)**;**
54. **for** (**int** v = 0**;** v <= 4**;** ++v) {
55. std**::**[cout](http://www.cplusplus.com/reference/iostream/cout.html) << "shortest path from 0->" << v << ": " << dist[v] << "\n"**;**
56. }
57. **return** 0**;**
58. }