به نام خدا

گزارش پروژه سوم درس شبکه های کامپیوتری

سارا رضائی منش ۸۱۰۱۹۸۵۷۶

برنا توسلی ۸۱۰۱۹۸۳۷۴

عنوان پروژه

آشنایی با الگوریتم های مسیریابی در شبکه

توضیحات فایل

کلاس NetWork:

```
class NetWork
{
public:
    void makeTopology(string topo);
    void showTopology();
    void lsrp(int src);
    void lsrpAll();
    void dvrp(int src);
    void dvrpAll();
    void removeLink(string input);
    void modifyLink(string input);

private:
    int getNextHop(int parent[], int j, int src);
    void printPath(int parent[], int j);
    void printPaths(int distance[], int parent[], int src);
    void printIteration(int dist[], int n, int parent[], int Iter);
    void printTable(int distance[], int parent[], int src);
    int findMinDistance(int dist[], bool visited[]);
    bool isPower(int x, long int y);
    vector<int> splitArgs(string args);

    int links[MAX_NODES][MAX_NODES];
    int n = 0;
};
```

این کلاس یک interface برای کار با شبکه ارائه می‌کند که با استفاده از آن دستورات صورت پروژه انجام شده و نتیجه آنها اعلام می‌شود.

۱) تابع makeTopology:

```
void NetWork::makeTopology(string topo) {
   stringstream ts(topo);
   string linkInfo;

   while(ts >> linkInfo) {
       vector<int> args = splitArgs(linkInfo);
       links[args[0]][args[1]] = links[args[1]][args[0]] = args[2];
       n = max(n, max(args[0]+1, args[1]+1));
   }
}
```

از این تابع برای اجرای دستور topology استفاده می‌شود. این تابع با گرفتن یک رشته شامل لینک ها و وزن هایشان، گراف شبکه را به صورت یک ماتریس مجاورت در آرایه دو بعدی links ذخیره می‌کند.

از آنجایی که نود ها از 1 تا n شماره گذاری شده اند، تعداد نود ها برابر با بزرگترین شماره نود دریافتی است که در حلقه while آپدیت می‌شود.

برای جدا کردن رشته ها به صورت u-v-w، از تابع splitArgs کمک گرفته شده است. این تابع با گرفتن رشته ذکر شده آن را پردازش و به یک آرایه به صورت [u, v, w] تبدیل کرده و بازمی‌گرداند.

۲) تابع showTopology:

```
void NetWork::showTopology() {
   cout << "u|v |  ";
   for(int i = 0; i < n; i++) {
       int spaces = i == 0 ? 2 : 3-log10(i+1);
       cout << " " * Multiplier(spaces) << i+1 << " " * Multiplier(spaces);
   }
   cout << endl << "-------" << "-----" * Multiplier(n) << endl;

   for(int i = 0; i < n*2; i++) {
       if(!(i%2) && i != 0) {
           cout << "    | \n";
           continue;
       } else if(i == 0) { continue; }
       string spaces = (i+1)/2 >= 10 ? " " : "  ";
       cout << " " << (i+1)/2 << spaces+"|";
       for(int j = 0; j < n; j++) {
           int spaces = 5-log10(links[i/2][j] >= 0 ? max(9, links[i/2][j]) : 12);
           cout << " " * Multiplier(spaces) << links[i/2][j];
       }
       cout << endl;
   }
}
```

از این تابع برای اجرای دستور show استفاده می‌شود. این تابع با استفاده از ماتریس مجاورت و یکسری عملیات پیچیده که توضیحات آنها قطعا از حوصله شما خارج است! توپولوژی شبکه را به فرمت توضیح داده شده در صورت پروژه و به صورت مرتب شده چاپ می‌کند.

نتیجه اجرای این عملیات بر روی مثال پایانی پروژه به صورت زیر است:

```
show
u|v |    1    2    3    4    5    6    7    8    9    10   11   12   13
-------------------------------------------------------------------------
 1  |    0   -1   -1   -1    6   -1    7   -1   -1   -1   -1   -1   -1
    |
 2  |   -1    0   -1   -1   -1    2   13   -1   -1   -1   -1   -1   -1
    |
 3  |   -1   -1    0   -1   -1   -1    1    4   -1   -1   -1    8   -1
    |
 4  |   -1   -1   -1    0   -1   19   -1   -1   -1   11   -1   -1   -1
    |
 5  |    6   -1   -1   -1    0   -1   -1   -1   -1    3   -1   -1   -1
    |
 6  |   -1    2   -1   19   -1    0   -1   17   -1   -1   25   -1    4
    |
 7  |    7   13    1   -1   -1   -1    0   -1   -1   -1   -1    8   -1
    |
 8  |   -1   -1    4   -1   -1   17   -1    0   -1   -1   16   -1   -1
    |
 9  |   -1   -1   -1   -1   -1   -1   -1   -1    0   -1   -1    5    7
    |
 10 |   -1   -1   -1   11    3   -1   -1   -1   -1    0   -1   12   -1
    |
 11 |   -1   -1   -1   -1   -1   25   -1   16   -1   -1    0   -1   -1
    |
 12 |   -1   -1    8   -1   -1   -1    8   -1    5   12   -1    0   -1
    |
 13 |   -1   -1   -1   -1   -1    4   -1   -1    7   -1   -1   -1    0
```

۱-۳) تابع lsrp:

```
void NetWork::lsrp(int src)
{
   auto start = high_resolution_clock::now();
   n = ceil(sqrt(nn*2));
   int distance[2000];
   bool visited[2000] = {false};
   int parent[2000] = {-1};

   for (int i = 0; i < n; i++)
       distance[i] = INT_MAX;
   distance[src] = 0;
   for (int iter = 0; iter < n-1; iter++) {
       int u = findMinDistance(distance, visited);
       visited[u] = true;
       for (int v = 0; v < n; v++)
           if (!visited[v] && links[u][v] != -1 && distance[u] != INT_MAX
               && distance[u] + links[u][v] < distance[v]) {
               parent[v] = u;
               distance[v] = distance[u] + links[u][v];
           }
       printIteration(distance, n, parent, iter+1);
   }
   parent[src] = -1;
   printPaths(distance, parent, src);
```

```
  auto end = high_resolution_clock::now();
   auto duration = duration_cast<microseconds>(end - start);
   cout << "time elapsed for executing link state algorithm for a single source = "
        << duration.count() << " microseconds" << endl;
}
```

از این تابع برای اجرای دستور lsrp برای یک مبدا استفاده می‌شود. الگوریتم link state همان الگوریتم dijkstra برای یافتن کوتاهترین مسیر می‌باشد. این الگوریتم به صورت زیر عمل می‌کند:

در الگوریتم link state در ابتدا یک آرایه visited، distance و parent ایجاد می‌کنیم که به ترتیب برای ذخیره وضعیت نود های گراف از نظر بررسی شدن در الگوریتم، فاصله نود مبدا از باقی گره ها و نود قبلی که در کوتاهترین مسیر از مبدا به نود فعلی از آن عبور کردیم مورد استفاده قرار می‌گیرند.

فاصله همه نود ها به جز مبدا تا مبدا، در شروع برابر با یک عدد بسیار بزرگ است و فاصله مبدا تا مبدا صفر می‌باشد.

سپس الگوریتم در هر مرحله نزدیکترین نود بررسی نشده(visited[i]= false) به مبدا را انتخاب می‌کند و در آرایه visited خانه مقدار مربوط به آن را true می‌کند که یعنی این نود بررسی شده است و در مراحل بعد نباید مجدد مورد بررسی قرار بگیرد. و فاصله مبدا را تا همسایه های خود محاسبه می‌کند. این فاصله همان فاصله نزدیکترین نود تا مبدا به اضافه فاصله لینک بین آن و نود همسایه اش است. در صورتی که مقدار محاسبه شده از فاصله فعلی که برای نود همسایه در آرایه distance ذخیره شده است کمتر باشد، یعنی مسیر کوتاهتری از مبدا به آن نود پیدا شده است. پس parent آن به نود فعلی مورد بررسی و فاصله آن به مقدار جدید آپدیت می‌شود. در پایان اجرای این الگوریتم، در distance مقدار کوتاهترین مسیر به هر نود از نود مبدا و در parent، برای هر نود، نود قبلی آن در مسیر ذخیره شده است.

برای نمایش دادن نتایج در هر پیمایش از تابع printIteration و برای نمایش دادن نتایج اجرای نتایج نهایی از تابع printPaths استفاده شده است.

ضمنا در تابع lsrp، برای محاسبه ی زمان همگرایی، در ابتدا و انتهای فانکشن، زمان حال دریافت شده و در انتها، مدت زمان اجرای تابع به میکرو ثانیه گزارش می‌شود.

خروجی این تابع برای یک iteration به صورت زیر است:

```
lsrp 1
Iter 1:
Dest   | 1| 2| 3| 4|
Cost   | 0| 19| 9| -1|
Iter 2:
Dest   | 1| 2| 3| 4|
Cost   | 0| 19| 9| -1|
Iter 3:
Dest   | 1| 2| 3| 4|
Cost   | 0| 19| 9| 22|
Path: [s] -> [d]     Min-Cost    Shortest Path
      ------------   ---------   -------------
      [1] --> [2]    19          1-->2
      [1] --> [3]    9           1-->3
      [1] --> [4]    22          1-->2-->4

time elapsed for executing link state algorithm for node 1 = 125 microseconds
```

۳-۲) تابع lsrpAll:

```
void NetWork::lsrpAll() {
   auto start = high_resolution_clock::now();
   n = ceil(sqrt(nn*2));
   for(int i = 1; i <= n; i++)
       lsrp(i-1);
   cout << endl;
   auto end = high_resolution_clock::now();
   auto duration = duration_cast<microseconds>(end - start);
   cout << "time elapsed for executing link state algorithm for all nodes = "
        << duration.count() << " microseconds" << endl;
}
```

این تابع همه ی نود های شبکه را یکبار مبدا در نظر گرفته و تابع lsrp را برای آنها اجرا می‌کند.

در تابع lsrpAll هم مشابه lsrp، برای محاسبه ی زمان همگرایی، در ابتدا و انتهای فانکشن، زمان حال دریافت شده و در انتها، مدت زمان اجرای تابع به میکرو ثانیه گزارش می‌شود.

۳-۳) تابع printIteration:

```
void NetWork::printIteration(int dist[], int n, int parent[], int Iter)
{
   cout << "Iter " << Iter << ":\nDest   |";


   for (int i = 1; i <= n; i++)
       cout << " " << i << "|";

   cout << "\nCost   |";
   for (int i = 1; i <= n; i++) {
       if(dist[i-1] == INT_MAX)
           cout << " -1|";
       else
           cout << " " << dist[i-1] << "|";
   }
   cout << endl;
}
```

این تابع با استفاده از آرایه های parent و iteration اطلاعات هر پیمایش را به فرمت گفته شده در صورت پروژه چاپ می‌کند.

۳-۴) تابع printPaths:

```
void NetWork::printPaths(int distance[],  int parent[], int src)
{
   cout << "Path: [s] -> [d]     Min-Cost    Shortest Path\n";
   cout << "      ------------   ---------   -------------";
   for (int i = 0; i < n; i++) {
```

```
        if(i == src)
            continue;

        distance[i] = distance[i] == INT_MAX ? -1 : distance[i];
        int spaces = 12 - log10(distance[i] >= 0 ? distance[i] : 12);
        int spaces2 = 4 - ((isPower(10, i+1)) ? log10(i+1) : log10(i+1)-1);
        cout << "\n      [" << src+1 << "] --> [" << i+1 << "]" + " " * Multiplier(spaces2) <<
distance[i] << " " * Multiplier(spaces);

        cout << src+1 << ARROW;

        printPath(parent, parent[i]);
        cout << i+1;
    }
    cout << endl << endl;
}
```

این تابع با استفاده از آرایه های parent و iteration نتایج کلی ران شدن الگوریتم link state را به فرمت گفته شده در صورت پروژه چاپ می‌کند. در این تابع به ترتیب مبدا و مقصد در هر مسیر، کمترین هزینه آن و در نهایت مسیری که برای رسیدن از مبدا به مقصد طی شده است چاپ می‌شود. برای چاپ کردن مسیر از تابع بازگشتی printPath کمک گرفته شده است. این تابع، در هر بار اجرا، برای مقدار ذخیره شده در ایندکس نود فعلی در آرایه parent مجدد فراخوانی می‌شود(یعنی نود قبلی در مسیر) و اینکار را تا جایی ادامه می‌دهد که به نودی برسد که مقدار پدرش -1 باشد. این یعنی به نود مبدا رسیده است و باید در بازگشت از آن نود، مسیر را چاپ کند تا به نود مقصد برسد.

۴-۱) تابع dvrp:

```
void NetWork::dvrp(int src)
{
    auto start = high_resolution_clock::now();
    int distance[MAX_NODES];
    int parent[MAX_NODES] = {-1};

    for (int i = 0; i < n; i++)
        distance[i] = INT_MAX;
    distance[src] = 0;
    for (int iter = 0; iter <= n - 1; iter++)
    {
        for (int i = 0; i < n; i++){
            for (int j = 0; j < n; j++){
                if (distance[i] != INT_MAX && links[i][j] != -1 && distance[i] + links[i][j] <
distance[j]){
                    distance[j] = distance[i] + links[i][j];
                    parent[j] = i;
                }
            }
        }

    }
    parent[src] = -1;
```

```
    printTable(distance, parent, src);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start);
    cout << "time elapsed for executing distance vector algorithm for node " << src + 1 << " =
"
         << duration.count() << " microseconds\n\n";
}
```

از این تابع برای اجرای دستور dvrp برای یک مبدا استفاده می‌شود. الگوریتم distance vector همان الگوریتم bellman-ford برای یافتن کوتاهترین مسیر می‌باشد. این الگوریتم به صورت زیر عمل می‌کند:

در الگوریتم distance vector در ابتدا یک آرایه distance و parent ایجاد می‌کنیم که به ترتیب برای ذخیره فاصله نود مبدا از باقی گره ها و نود قبلی که در کوتاهترین مسیر از مبدا به نود فعلی از آن عبور کردیم مورد استفاده قرار می‌گیرند.

فاصله همه نود ها به جز مبدا تا مبدا، در شروع برابر با یک عدد بسیار بزرگ است و فاصله مبدا تا مبدا صفر می‌باشد.

سپس الگوریتم با n بار جرخش روی تمامی یالها درصورتی که مسیر کوتاه‌تری به یک راس (با شروع از راس مبدا) پیدا کند، مسیر جدید را جایگزین مسیر قبلی می‌کند. پس parent آن به نود فعلی مورد بررسی و فاصله آن به مقدار جدید آپدیت می‌شود. در پایان اجرای این الگوریتم، در distance مقدار کوتاهترین مسیر به هر نود از نود مبدا و در parent، برای هر نود، نود قبلی آن در مسیر ذخیره شده است.

برای نمایش دادن نتایج اجرای نتایج نهایی از تابع printTable استفاده شده است.

ضمنا در تابع dvrp، برای محاسبه ی زمان همگرایی، در ابتدا و انتهای فانکشن، زمان حال دریافت شده و در انتها، مدت زمان اجرای تابع به میکرو ثانیه گزارش می‌شود.

خروجی این تابع برای یک iteration به صورت زیر است:

```
dvrp 1
Dest            Next Hop        Dist            Shortest Path
-------------------------------------------------------------
1               1               0               [1]
2               2               19              [1-->2]
3               3               9               [1-->3]
4               2               22              [1-->2-->4]

time elapsed for executing distance vector algorithm for node 1 = 182 microseconds
```

۴-۲) تابع dvrpAll:

```
void NetWork::dvrpAll()
{
    auto start = high_resolution_clock::now();
```

```
    for (int i = 1; i <= n; i++)
        dvrp(i - 1);
    cout << endl;
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start);
    cout << "time elapsed for executing distance vector algorithm for all nodes = "
         << duration.count() << " microseconds\n\n";
}
```

این تابع همه ی نود های شبکه را یکبار مبدا در نظر گرفته و تابع dvrp را برای آنها اجرا می‌کند.

در تابع dvrpAll هم مشابه dvrp، برای محاسبه ی زمان همگرایی، در ابتدا و انتهای فانکشن، زمان حال دریافت شده و در انتها، مدت زمان اجرای تابع به میکرو ثانیه گزارش می‌شود.

۳-۴) تابع printTable:

```
void NetWork::printTable(int distance[], int parent[], int src)
{
    int col_space = 15;
    cout << "Dest           Next Hop       Dist           Shortest Path  \n";
    cout << "-------------------------------------------------------------------\n";

    for (int i = 0; i < n; i++)
    {
        int next_hop = getNextHop(parent, i, src);
        cout << i+1;
        cout << string(col_space - to_string(i+1).size(), ' ');
        cout << next_hop + 1;
        cout << string(col_space - to_string(next_hop + 1).size(), ' ');
        cout << distance[i];
        cout << string(col_space - to_string(distance[i]).size(), ' ');
        cout << "[";
        cout << src + 1;
        if (src != i){
            cout << ARROW;
            printPath(parent, parent[i]);
            cout << i + 1;
        }
        cout << "]\n";

    }
    cout << endl;
}
```

این تابع با استفاده از آرایه فاصله‌ها، وراثت و راس مبدا و همچنین با فراخوانی تابع printPath اقدام به چاپ نتایج فواصل نهایی برای راس مبدا src می‌کند. دقت کنید که next hop با استفاده از تابعی جدا محاسبه و شماره راس آن برگردانده شده؛ همچنین اگر راس مبدا با مقصد برابر بود، دیگر نیازی به فراخوانی printPath نداریم.

۴-۴) تابع getNextHop:

```
int NetWork::getNextHop(int parent[], int j, int src)
{
    if (parent[j] == src)
        return j;
    if (parent[j] == -1)
        return j;
    return getNextHop(parent, parent[j], src);
}
```

این تابع با استفاده از آرایه وراثت اولین پرش ما بعد از راس src را محاسبه کرده و شماره آن را باز می‌گرداند.

۵) تابع removeLink:

```
void NetWork::removeLink(string input) {
   vector<int> args = splitArgs(input);
   links[args[0]][args[1]] = links[args[1]][args[0]] = -1;
}
```

از این تابع برای اجرای دستور remove استفاده می‌شود. برای حذف لینک بین دو نود کافی است فاصله بین آن نود ها را در ماتریس مجاورت برابر با -1 قرار دهیم. در این تابع ابتدا با استفاده از تابع splitArgs، دو نودی که می‌خواهیم لینک بین آنها را حذف کنیم بدست آورده و سپس ماتریس مجاورت را آپدیت می‌کنیم.(از آنجایی که مسیر ها بدون جهت هستند باید مقدار دو خانه آپدیت شود.)

۶) تابع modifyLink:

```
void NetWork::modifyLink(string input) {
   vector<int> args = splitArgs(input);
   if(args[0] == args[1]) {
       cout << "can't modify the distance of a node from itself!";
       return;
```

```
    }
    links[args[0]][args[1]] = links[args[1]][args[0]] = args[2];
}
```

از این تابع برای اجرای دستور modify استفاده می‌شود. برای تغییر لینک بین دو نود کافی است فاصله بین آن نود ها را در ماتریس مجاورت به مقدار داده شده تغییر دهیم. در این تابع ابتدا با استفاده از تابع splitArgs، دو نودی که می‌خواهیم لینک بین آنها را حذف کنیم بدست آورده و سپس ماتریس مجاورت را آپدیت می‌کنیم.(از آنجایی که مسیر ها بدون جهت هستند باید مقدار دو خانه آپدیت شود.)

در صورتی که دو نودی که به عنوان آرگومان های این دستور با هم یکی باشند، خطای مربوطه چاپ شده و اجرای تابع متوقف می‌شود.

۷) تابع handleCommand:

```
void handleCommands(NetWork *network)
{
    string command, args;
    while (cin >> command)
    {
        getline(cin, args);
        if (command == "topology")
            network->makeTopology(args);
        if (command == "show")
            network->showTopology();
        if (command == "lsrp")
            (args != "") ? network->lsrp(stoi(args) - 1) : network->lsrpAll();
        if(command == "dvrp")
            (args != "") ? network->dvrp(stoi(args) - 1) : network->dvrpAll();
        if (command == "remove")
            network->removeLink(args);
        if (command == "modify")
            network->modifyLink(args);
    }
}
```

این تابع برای پردازش و ارسال دستورها به توابع کلاس NetWork مورد استفاده قرار می‌گیرد. handleCommand در ابتدا دستور و آرگومان آن را(در صورت وجود) دریافت کرده و با توجه به نوع دستور، تابع متناظر آن را فراخوانی می‌کند. از آنجایی که دستورهای lsrp و dvrp می‌توانند بدون آرگومان وارد شوند، در صورتی که مقدار args پس از خوانده شدن از ورودی خالی بود، lsrp و dvrp نتایج را برای تمامی گره ها چاپ می‌کنند.

مقایسه نتایج

الگوریتم link state:

برای هر نود در الگوریتم link state، زمان های ثبت شده به شرح زیر هستند:

```
lsrp
time elapsed for executing link state algorithm for node 1 = 11 microseconds

time elapsed for executing link state algorithm for node 2 = 11 microseconds

time elapsed for executing link state algorithm for node 3 = 10 microseconds

time elapsed for executing link state algorithm for node 4 = 10 microseconds

time elapsed for executing link state algorithm for node 5 = 9 microseconds

time elapsed for executing link state algorithm for node 6 = 8 microseconds

time elapsed for executing link state algorithm for node 7 = 8 microseconds

time elapsed for executing link state algorithm for node 8 = 8 microseconds

time elapsed for executing link state algorithm for node 9 = 9 microseconds

time elapsed for executing link state algorithm for node 10 = 9 microseconds

time elapsed for executing link state algorithm for node 11 = 9 microseconds

time elapsed for executing link state algorithm for node 12 = 9 microseconds

time elapsed for executing link state algorithm for node 13 = 8 microseconds
```

و زمان ثبت شده برای اجرای الگوریتم بر روی کل نود ها به صورت زیر است:

```
time elapsed for executing link state algorithm for all nodes = 280 microseconds
```

نتایج نهایی مسیریابی از نود چهارم توسط این الگوریتم به صورت زیر است:

```
Path: [s] -> [d]     Min-Cost    Shortest Path
      ------------   ---------   -------------
      [4] --> [1]    20          4-->10-->5-->1
      [4] --> [2]    21          4-->6-->2
      [4] --> [3]    28          4-->10-->5-->1-->7-->3
      [4] --> [5]    14          4-->10-->5
      [4] --> [6]    19          4-->6
      [4] --> [7]    27          4-->10-->5-->1-->7
      [4] --> [8]    32          4-->10-->5-->1-->7-->3-->8
      [4] --> [9]    28          4-->10-->12-->9
      [4] --> [10]   11          4-->10
      [4] --> [11]   44          4-->6-->11
      [4] --> [12]   23          4-->10-->12
      [4] --> [13]   23          4-->6-->13
```

پس از حذف لینک بین نود های ۱۰ و ۴، زمان ثبت شده برای مسیریابی های مجدد به صورت زیر است:

```
lsrp
time elapsed for executing link state algorithm for node 1 = 14 microseconds

time elapsed for executing link state algorithm for node 2 = 15 microseconds

time elapsed for executing link state algorithm for node 3 = 14 microseconds

time elapsed for executing link state algorithm for node 4 = 15 microseconds

time elapsed for executing link state algorithm for node 5 = 13 microseconds

time elapsed for executing link state algorithm for node 6 = 13 microseconds

time elapsed for executing link state algorithm for node 7 = 12 microseconds

time elapsed for executing link state algorithm for node 8 = 13 microseconds

time elapsed for executing link state algorithm for node 9 = 13 microseconds

time elapsed for executing link state algorithm for node 10 = 13 microseconds

time elapsed for executing link state algorithm for node 11 = 12 microseconds

time elapsed for executing link state algorithm for node 12 = 11 microseconds

time elapsed for executing link state algorithm for node 13 = 13 microseconds


time elapsed for executing link state algorithm for all nodes = 321 microseconds
```

نتایج مسیریابی از نود اول توسط این الگوریتم بعد از حذف لینک بین نود های ۴ و ۱۰ به صورت زیر است:

```
Path: [s] -> [d]     Min-Cost    Shortest Path
      ------------   ---------   -------------
      [4] --> [1]    41          4-->6-->2-->7-->1
      [4] --> [2]    21          4-->6-->2
      [4] --> [3]    35          4-->6-->2-->7-->3
      [4] --> [5]    47          4-->6-->2-->7-->1-->5
      [4] --> [6]    19          4-->6
      [4] --> [7]    34          4-->6-->2-->7
      [4] --> [8]    36          4-->6-->8
      [4] --> [9]    30          4-->6-->13-->9
      [4] --> [10]   47          4-->6-->13-->9-->12-->10
      [4] --> [11]   44          4-->6-->11
      [4] --> [12]   35          4-->6-->13-->9-->12
      [4] --> [13]   23          4-->6-->13
```

مشاهده می‌شود که با حذف لینک بین نود های چهار و ده، طول برخی مسیر ها طولانی تر شده است.

الگوریتم distance vector:

برای هر نود در الگوریتم distance vector، زمان های ثبت شده به شرح زیر هستند:

```
dvrp
time elapsed for executing distance vector algorithm for node 1 = 23 microseconds

time elapsed for executing distance vector algorithm for node 2 = 27 microseconds

time elapsed for executing distance vector algorithm for node 3 = 18 microseconds

time elapsed for executing distance vector algorithm for node 4 = 16 microseconds

time elapsed for executing distance vector algorithm for node 5 = 17 microseconds

time elapsed for executing distance vector algorithm for node 6 = 15 microseconds

time elapsed for executing distance vector algorithm for node 7 = 17 microseconds

time elapsed for executing distance vector algorithm for node 8 = 15 microseconds

time elapsed for executing distance vector algorithm for node 9 = 16 microseconds

time elapsed for executing distance vector algorithm for node 10 = 16 microseconds

time elapsed for executing distance vector algorithm for node 11 = 17 microseconds

time elapsed for executing distance vector algorithm for node 12 = 17 microseconds

time elapsed for executing distance vector algorithm for node 13 = 16 microseconds

time elapsed for executing distance vector algorithm for all nodes = 414 microseconds
```

نتایج نهایی مسیریابی از نود چهارم توسط این الگوریتم به صورت زیر است:

```
dvrp 4
Dest            Next Hop        Dist            Shortest Path
-------------------------------------------------------------
1               10              20              [4-->10-->5-->1]
2               6               21              [4-->6-->2]
3               10              28              [4-->10-->5-->1-->7-->3]
4               4               0               [4]
5               10              14              [4-->10-->5]
6               6               19              [4-->6]
7               10              27              [4-->10-->5-->1-->7]
8               10              32              [4-->10-->5-->1-->7-->3-->8]
9               10              28              [4-->10-->12-->9]
10              10              11              [4-->10]
11              6               44              [4-->6-->11]
12              10              23              [4-->10-->12]
13              6               23              [4-->6-->13]

time elapsed for executing distance vector algorithm for node 4 = 273 microseconds
```

پس از حذف لینک بین نود های ۱۰ و ۴، زمان ثبت شده برای مسیریابی های مجدد به صورت زیر است:

```
dvrp
time elapsed for executing distance vector algorithm for node 1 = 32 microseconds

time elapsed for executing distance vector algorithm for node 2 = 27 microseconds

time elapsed for executing distance vector algorithm for node 3 = 22 microseconds

time elapsed for executing distance vector algorithm for node 4 = 16 microseconds

time elapsed for executing distance vector algorithm for node 5 = 17 microseconds

time elapsed for executing distance vector algorithm for node 6 = 15 microseconds

time elapsed for executing distance vector algorithm for node 7 = 25 microseconds

time elapsed for executing distance vector algorithm for node 8 = 30 microseconds

time elapsed for executing distance vector algorithm for node 9 = 20 microseconds

time elapsed for executing distance vector algorithm for node 10 = 16 microseconds

time elapsed for executing distance vector algorithm for node 11 = 15 microseconds

time elapsed for executing distance vector algorithm for node 12 = 15 microseconds

time elapsed for executing distance vector algorithm for node 13 = 15 microseconds


time elapsed for executing distance vector algorithm for all nodes = 423 microseconds
```

نتایج مسیریابی از نود اول توسط این الگوریتم بعد از حذف لینک بین نود های ۴ و ۱۰ به صورت زیر است:

```
dvrp 4
Dest            Next Hop        Dist            Shortest Path
-------------------------------------------------------------------
1               6               41              [4-->6-->2-->7-->1]
2               6               21              [4-->6-->2]
3               6               35              [4-->6-->2-->7-->3]
4               4               0               [4]
5               6               47              [4-->6-->2-->7-->1-->5]
6               6               19              [4-->6]
7               6               34              [4-->6-->2-->7]
8               6               36              [4-->6-->8]
9               6               30              [4-->6-->13-->9]
10              6               47              [4-->6-->13-->9-->12-->10]
11              6               44              [4-->6-->11]
12              6               35              [4-->6-->13-->9-->12]
13              6               23              [4-->6-->13]

time elapsed for executing distance vector algorithm for node 4 = 296 microseconds
```

مشاهده می‌شود که با حذف لینک بین نود های چهار و ده، طول برخی مسیر ها طولانی تر شده است.

مقایسه دو روش

زمان اجرا: از آنجایی که پیچیدگی زمانی الگوریتم بلمن-فورد از دایکسترا بیشتر است، به طور میانگین اجرای الگوریتم distance vector زمان بیشتری برده است.

مسیریابی: همانطور که در شکل های بالا مشاهده می‌شود، مسیرهای پیدا شده از نود چهار و همچنین نود های دیگر به سایر نود ها، توسط هر دو الگوریتم بهینه بوده و یکسان هستند.