definicja

Graf prosty, którego wierzchołkami są wszystkie k-elementowe ciągi binarne i w którym krawędzie łączą tylko te spośród ciągów, które różnią się dokładnie jednym elementem, nazywamy k-kostką (hiperkostką) i oznaczamy H_k

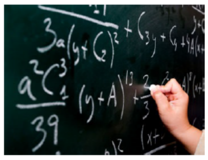
Hiperkostaka H_1 składa się z dwóch wierzhołków połączonych krawędzią

Idea algorytmu

- algorytm wykrywa ścieżki powiększające, dopóki istnieją
- zwiększa przepływ po tych ścieżkach o ich pojemność

```
algorithm augmenting path;
begin
    x : = 0;
    while G(x) contains a directed path from node s to node t do
    begin
        identify an augmenting path P from node s to node t;
        δ : = min{r_ij : (i, j) ∈ P};
        augment δ units of flow along P and update G(x);
    end;
end;
```

Przykład



Uwagi

- algorytm znany jako algorytm/metoda Forda-Fulkersona
- istnieje szereg implementacji o różnych złożonościach

oznaczaja¸cego wymiar hiperkostki,
wygeneruje opisany wy˙zej graf skierowany H_k oraz obliczy maksymalny przepływ mi¸edzy ´zródłem $s = 0$ a uj´sciem $t = 2k - 1$.

- The Ford–Fulkerson algorithm is essentially a greedy algorithm. If there are multiple possible augmenting paths, the decision of which path to use in line 2 is completely arbitrary.[2] Thus, like any terminating greedy algorithm, the Ford–Fulkerson algorithm will find a locally optimal solution; it remains to show that the local optimum is also a global optimum. This is done in §13.2.

Now that we have laid out the necessary conceptual machinery, let's give more detailed pseudocode for the Ford–Fulkerson algorithm.

Algorithm: FORD–FULKERSON(G)

$\triangleright$ Initialize flow f to zero
for each edge $(u,v) \in E$ **do**
    $(u,v).f \leftarrow 0$
$\triangleright$ The following line runs a graph search algorithm (such as BFS or DFS)* to find a path from s to t in G_f
while there exists a path $p : s \rightsquigarrow t$ in G_f **do**
    $c_f(p) \leftarrow \min\{c_f(u,v) : (u,v) \in p\}$
    for each edge $(u,v) \in p$ **do**
        $\triangleright$ Because $(u,v) \in G_f$, it must be the case that either $(u,v) \in E$ or $(v,u) \in E$.
        $\triangleright$ And since G is a flow network, the "or" is exclusive: $(u,v) \in E$ xor $(v,u) \in E$.
        if $(u,v) \in E$ **then**
            $(u,v).f \leftarrow (u,v).f + c_f(p)$
        else
            $(v,u).f \leftarrow (v,u).f - c_f(p)$

* For more information about breath-first and depth-first searches, see Sections 22.2 and 22.3 of CLRS.

Edmonds-Karp is a specialisation/elaboration of Ford-Fulkerson, so any bound for the latter also applies to the former. In other words, EK is $O(|E| \min(f_{max}, |V||E|))$ time (and writing it this way does add information, since f_{max} can be much smaller than $|V||E|$ -- and this is the only time when you might otherwise consider using some other variant of FF in preference to EK).

Nasz algorytm:

Prerequisite : **Max Flow Problem Introduction**

```
Ford-Fulkerson Algorithm
The following is simple idea of Ford-Fulkerson algorithm:
1) Start with initial flow as 0.
2) While there is a augmenting path from source to sink.
           Add this path-flow to flow.
3) Return flow.
```

Time Complexity: Time complexity of the above algorithm is O(max_flow * E). We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max_flow * E).

The above implementation of Ford Fulkerson Algorithm is called **Edmonds-Karp Algorithm**. The idea of Edmonds-Karp is to use BFS in Ford Fulkerson implementation as BFS always picks a path with minimum number of edges. When BFS is used, the worst case time complexity can be reduced to $O(VE^2)$. The above implementation uses adjacency matrix representation though where BFS takes $O(V^2)$ time, the time complexity of the above implementation is $O(EV^3)$ (Refer CLRS book for proof of time complexity)

Macierz sąsiedztwa

BFS:

Input: The vertices list, the start node, and the sink node.

```
Begin
   initially mark all nodes as unvisited
   state of start as visited
```

```
   predecessor of start node is φ
   insert start into the queue qu
   while qu is not empty, do
      delete element from queue and set to vertex u
      for all vertices i, in the residual graph, do
         if u and i are connected, and i is unvisited, then
            add vertex i into the queue
            predecessor of i is u
            mark i as visited
      done
   done
   return true if state of sink vertex is visited
End
```

FORD-FULKERSON:

Input: The vertices list, the source vertex, and the sink vertex.

Output – The maximum flow from start to sink.

```
Begin
   create a residual graph and copy given graph into it
   while bfs(vert, source, sink) is true, do
      pathFlow := ∞
      v := sink vertex
      while v ≠ start vertex, do
         u := predecessor of v (poprzednik)
         pathFlow := minimum of pathFlow and residualGraph[u, v]
         v := predecessor of v
      done

      v := sink vertex
      while v ≠ start vertex, do
         u := predecessor of v
         residualGraph[u,v] := residualGraph[u,v] – pathFlow
         residualGraph[v,u] := residualGraph[v,u] – pathFlow
         v := predecessor of v
      done

      maFlow := maxFlow + pathFlow
   done
```

```
    return maxFlow
End
```