GRAFOVI

Rekurzivni obilazak grafa po dubini

- Odrediti broj čvorova grafa koji se mogu obići iz zadataog čvora
- Koristiti obilazak grafa po dubini
- Obilazak implementirati rekurzivno

Rekurzivni obilazak grafa po dubini

```
template <class T, class W>
long GraphAsLists<T,W>::dfs(T first)
{
  LinkedNode<T,W>* pStart;
  LinkedNode<T,W> *ptr = start;
  while(ptr != NULL){
      ptr->status = 0;
      if (ptr->node == first) {
            pStart = ptr;
      }
      ptr = ptr->next;
  }
  return dfs(pStart);
}
```

Rekurzivni obilazak grafa po dubini

```
template <class T, class W>
long GraphAsLists<T,W>::dfs(LinkedNode<T,W>* ptr)
{
   int retVal = 0;

   if (ptr->status != 1) {
        ptr->status = 1;
        Edge<T,W>* pot = ptr->adj;
        while(pot != NULL){
               retVal += dfs(pot->dest);
               pot = pot->link;
        }
        retVal++;
   }
   return retVal;
}
```

Rekurzivno određivanje puta između dva čvora obilaskom po dubini

- Odrediti put između dva čvora u grafu
  - Put čine svi čvorovi kroz koje treba proći da bi se stiglo iz jednog u drugi čvor
- Koristiti obilazak grafa po dubini
- Obilazak implementirati rekurzivno

Rekurzivno određivanje puta između dva čvora obilaskom po dubini

```
template <class T, class W> long GraphAsLists<T,W>::findPath
   (T first, T last, T *arPath/*=NULL*/, int *lPath/*=NULL*/)
{
   LinkedNode<T,W> *pStart, *pEnd;
   LinkedNode<T,W> *ptr = start;
   while(ptr != NULL){
        ptr->status = 0;
        if (ptr->node == first) {
               pStart = ptr;
        }
        if (ptr->node == last) {
               pEnd = ptr;
        }
        ptr = ptr->next;
   }
   return findPath(pStart, pEnd, arPath, lPath);
}
```

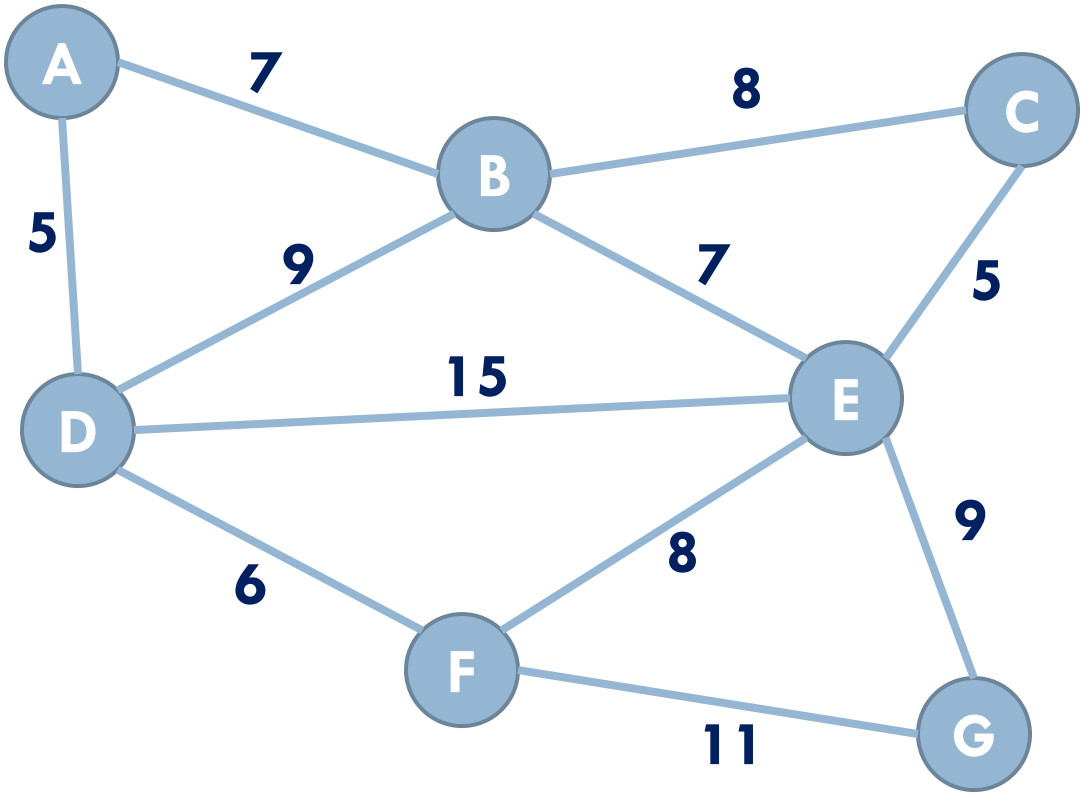
Rekurzivno određivanje puta između dva čvora obilaskom po dubini

```
template <class T, class W>
long GraphAsLists<T,W>::findPath(LinkedNode<T,W>* ptr,
        LinkedNode<T,W>* pEnd, T *arPath/*=NULL*/, int *lPath/*=NULL*/)
{
   if (ptr->node == pEnd->node) {
        if (arPath != NULL)
                arPath[*lPath++] = ptr->node;
        return 1;
   }
   int retVal = 0;
   if (ptr->status != 1) {
        ptr->status = 1;
        Edge<T,W>* pot = ptr->adj;
        while(retVal == 0 && pot != NULL) {
                retVal = findPath(pot->dest, pEnd, arPath, lPath);
                pot = pot->link;
        }
        if (retVal != 0 && arPath != NULL)
                arPath[*lPath++] = ptr->node;
   }
   return retVal;
}
```

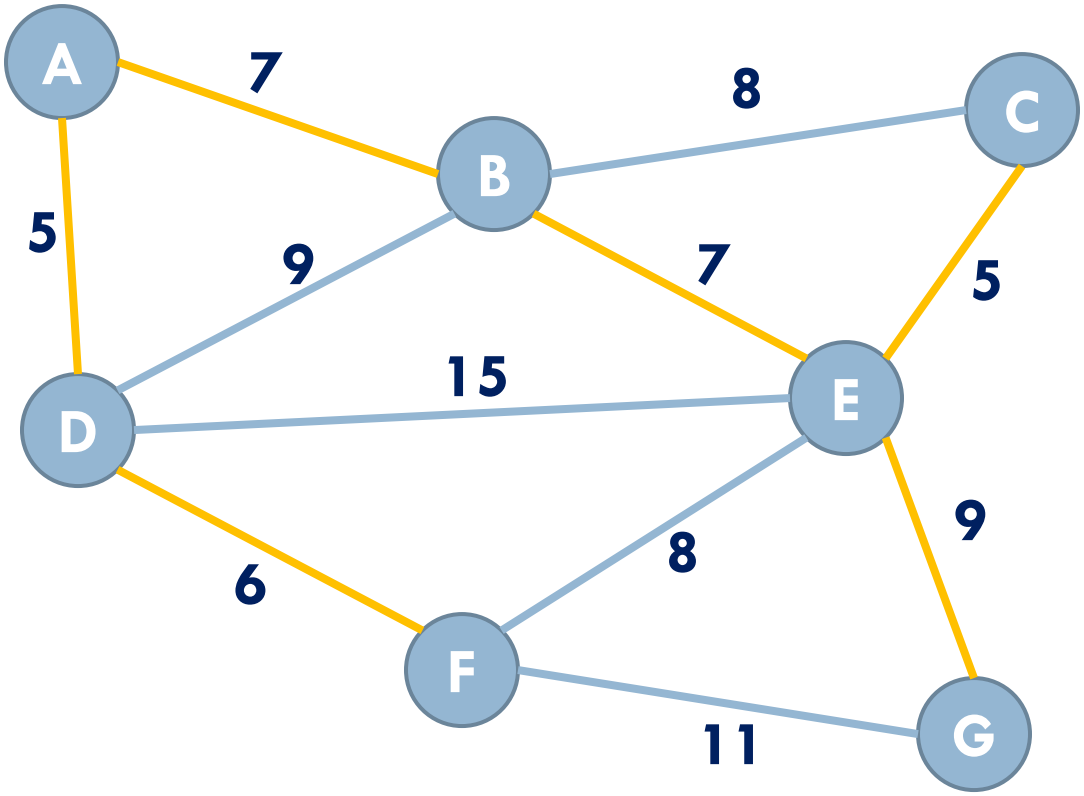
Minimalno sprežno stablo

- Kruskal-ov metod
- Prim-ov metod
- Reverse-Delete metod

Minimalno sprežno stablo



Minimalno sprežno stablo



Prim-ov metod

- Preduslovi:
  - Neprazan neusmeren povezan težinski graf
- Inicijalizacija:
  - Izabrati proizvoljan čvor grafa za početni i dodati ga u MST
- Sve dok broj čvorova MST nije jednak broju čvorova grafa
  - Izabrati granu sa minimalnom težinom, tako da je samo jedan njen čvor u MST, a drugi ne
  - Dodati ovu granu u MST

Prim-ov metod

```
template <class T, class W>
long GraphAsLists<T,W>::prim(GraphAsLists<T, W> &graph)
{
   LinkedNode<T, W> *ptr;
   Edge<T, W> *pot;
   SLList< LinkedEdge<T, W> > llist;

   ptr = start;
   while (ptr != NULL) {
        pot = ptr->adj;
        while (pot != NULL) {
                LinkedEdge<T, W> edge(ptr, pot->dest, pot->weight);
                SLLNode< LinkedEdge<T, W> > *pNode =
                                new SLLNode< LinkedEdge<T, W> >(edge);
                llist.InsertInSorted(pNode);
                pot = pot->link;
        }
        ptr = ptr->next;
   }
```

Prim-ov metod

```
graph.insertNode(start->node);
while (graph.nodeNum != nodeNum) {
	bool bOk = false;
	bool bInSrc, bInDest;
	SLLNode< LinkedEdge<T, W> > *curr, *next = NULL;
	curr = llist.getTail();
	while (!bOk && curr!=NULL) {
		bInSrc = graph.findNode(curr->info.src->node) != NULL;
		bInDest = graph.findNode(curr->info.dest->node) != NULL;
		bOk = bInSrc && !bInDest || !bInSrc && bInDest;
		if (bInSrc && bInDest) {
			llist.deleteEl(curr->info);
			if (next != NULL) {
				curr = llist.getPrev(next);
			} else {
				curr = llist.getTail();
			}
		} else {
			next = curr;
			curr = llist.getPrev(curr);
		}
	}
```

Prim-ov metod

```
        if (bOk) {
                if (bInSrc) {
                        graph.insertNode(next->info.dest->node);
                } else {
                        graph.insertNode(next->info.src->node);
                }
                graph.insertEdge(next->info.src->node,
                      next->info.dest->node, next->info.weight);
                graph.insertEdge(next->info.dest->node,
                      next->info.src->node, next->info.weight);
                llist.deleteEl(next->info);
        }
    }

    return true;
}
```

Reverse-Delete metod

- Preduslovi:
  - Neusmeren težinski graf
- Formirati sortiran niz grana u opadajući redosled po težinama
- Za svaku granu u nizu
  - Obrisati granu
  - Ako su čvorovi koje povezuje ova grana idalje povezani
    - Obrisati granu iz grafa

Reverse-Delete metod

```
template <class T, class W>
long GraphAsLists<T,W>::reverseDelete()
{
   LinkedNode<T, W> *ptr;
   Edge<T, W> *pot;
   SLList< LinkedEdge<T, W> > llist;

   ptr = start;
   while (ptr != NULL) {
        pot = ptr->adj;
        while (pot != NULL) {
                LinkedEdge<T, W> edge(ptr, pot->dest, pot->weight);
                SLLNode< LinkedEdge<T, W> > *pNode =
                                new SLLNode< LinkedEdge<T, W> >(edge);
                llist.InsertInSorted(pNode);
                pot = pot->link;
        }
        ptr = ptr->next;
   }
```

Reverse-Delete metod

```
    LinkedEdge<T, W> edge;
    LinkedNode<T, W> *ptr1, *ptr2;
    while (!llist.isEmpty()) {

        edge = llist.deleteFromHead();
        ptr1 = edge.src;
        ptr2 = edge.dest;

        deleteEdge(ptr1->node, ptr2->node);
        deleteEdge(ptr2->node, ptr1->node);
        if (findPath(ptr1->node, ptr2->node) == 0) {
                insertEdge(ptr1->node, ptr2->node, edge.weight);
                insertEdge(ptr2->node, ptr1->node, edge.weight);
        }
    }

    return true;
}
```