SPANNING FOREST TEMPLATE

Concept Spanning_Forest_Template(**type** Edge_Label;
eval Vertex_Max, Max_Edge_Count: Integer);
uses Std_Integer_Fac, Std_Boolean_Fac, Std_Real_Num_Fac;
requires Vertex_Max ≥ 1 **and** Max_Edge_Count ≥ 1 **which_entails**
Max_Edge_Count: $\mathbb{N}^{>0}$;

Def. **const** Vertex: $\wp(\mathbb{Z})$ = [1..Vertex_Max];

Def. **const** Edge: $\mathbb{S}\mathbb{e}\mathbb{t}$ = **Cart_Prod**
Ends: { S: $\wp$(Vertex) | 1 ≤ ||S|| ≤ 2 };
Cost: $\mathbb{R}^{>0}$;
Label: Edge_Label;
Inst_Num: [1..Max_Edge_Count]
end;

Family Graph_Holder ⊆ **Cart_Prod**
Gr_Edges: $\wp$(Edge);
Span_Found: $\mathbb{B}$
end;
exemplar GH;
Def const Edge_Count(GH: Graph_Holder): $\mathbb{C}\mathbb{a}\mathbb{r}\mathbb{d}$ = (||{ GH.Gr_Edges }||);
constraint Edge_Count(GH) ≤ Max_Edge_Count **which_entails** Edge_Count(GH): $\mathbb{N}$;
initialization
ensures ¬ GH.Span_Found **and** Edge_Count(GH) = 0;

Def const Are_Connected_in(GH: Graph_Holder; v, w: Vertex): $\mathbb{B}$ = (
∃ ρ: Str(Vertex) ∍ ⟨v⟩ Is_Prefix ρ **and** ⟨w⟩ Is_Suffix ρ **and** ∀ x, y: Vertex,
if ⟨x⟩∘⟨y⟩ Is_Substring ρ, **then** {x, y} ∈ GH.Gr_Edges);

Def const (GH1: Graph_Holder)Has_Same_Conn(GH2: Graph_Holder): $\mathbb{B}$ = (
∀ v, w: Vertex, Are_Connected_in(GH1, v, w) **iff** Are_Connected_in(GH2, v, w));

Def const (SGH: Graph_Holder)Is_Subgraph(GH: Graph_Holder): $\mathbb{B}$ = (
SGH.Gr_Edges ⊆ GH.Gr_Edges);

Def const (MGH: Graph_Holder)Is_Min_Span(GH: Graph_Holder): $\mathbb{B}$ = (
(MGH)Is_Subgraph(GH) **and** (MGH)Has_Same_Conn(GH) **and** ∀ GH3: Graph_Holder,
if (GH3)Is_Subgraph(GH) **and** (GH3)Has_Same_Conn(GH),
then $\sum_{E:\,GH3.Gr\_Edges}$ E.Cost ≥ $\sum_{E:\,MGH.Gr\_Edges}$ E.Cost);

```
    Def const ( SGH: Graph_Holder )Retains_Min_Span( GH: Graph_Holder ): 𝔹 = (
        (SGH)Is_Subgraph(GH) and ∃ MGH: Graph_Holder ∍
                                    (MGH)Is_Subgraph( SGH) and (MGH)Is_Min_Span(GH)  );

    Oper Augment_w( eval V1, V2: Integer; eval C: Real; alt L: Edge_Label;
                                        upd GH: Graph_Holder, aux IN: [1..Max_Edge_Count] );
        requires ¬ GH.Span_Found and Edge_Count(GH) + 1 ≤ Max_Edge_Count and C > 0.0
            and 1 ≤ V1 ≤ Vertex_Max and 1 ≤ V2 ≤ Vertex_Max which_entails V1, V2: Vertex;
        ensures ¬ GH.Span_Found and ⟨{V1, V2}, C, @L, IN⟩ ∉ @GH.Gr_Edges and
            ( GH )Retains_Min_Span( ⟨@GH.Gr_Edges ⋃ {⟨{V1, V2}, C, @L, IN⟩},
                                                                @GH.Span_Found⟩ );

    Oper Find_Min_Span( upd GH: Graph_Holder );
        requires ¬ GH.Span_Found;
        ensures GH.Span_Found and (GH)Is_Min_Span(@GH);

    Oper Remove_an_Edge( rpl V1, V2: Integer; rpl C: Real; rpl L: Edge_Label;
                                        upd GH: Graph_Holder, aux IN: Edge_Universe );
        requires Edge_Count(GH) ≥ 1;
        ensures @GH = ⟨GH.Gr_Edges ⋃ {⟨{V1, V2}, C, L, IN⟩}, GH.Span_Found⟩ and
                                                    ⟨{V1, V2}, C, L, IN⟩ ∉ GH.Gr_Edges;
    Oper Edge_Ct( rest GH: Graph_Holder ): Integer;
        ensures Edge_Ct = ( Edge_Count(GH) );

    Oper Span_Is_Found( rest GH: Graph_Holder ): Boolean;
        ensures Span_Is_Found = ( GH.Span_Found );

    Oper Clear_Graph( clr GH: Graph_Holder );

end Spanning_Forest_Template;
```

KRUSKAL REALIZATION

Realization Kruskal_Realiz(**realization** Arb_Pri_Realiz(
Oper Is_Leq(**rest** x, y: Entry): Boolean;
ensures Is_Leq = (x ⊴ y);)
for Prioritizer_Template;
realization Arb_Equiv_Realiz
for Coalescable_Equivalence_Relation_Template)
for Spanning_Forest_Template;
uses Prioritizer_Template, Coalescable_Equivalence_Relation_Template,
Standard_Cart_Prod_Realiz;

Facility Edge_Record_Fac;
uses Std_Integer_Fac, Std_Real_Fac, Edge_Label;

Type Edge_Data_Rec = **Record**
Vert1, Vert2: Vertex;
Cst: Real;
Lab: Edge_Label
end;
exemplar ED;
constraints ED.Cst > 0.0 **and** 1 ≤ ED.Vert1 ≤ Vertex_Max **and**
1 ≤ ED.Vert2 ≤ Vertex_Max;
initialization
ensures ED.Cst = 1.0 **and** ED.Vert1 = ED.Vert2 = 1;
procedure
ED.Cst := 1.0;
ED.Vert1 := 1;
ED.Vert2 := 1;
end initialization;

Oper Form_Edge_Record(**eval** V1, V2: Integer; **eval** C: Real; **alt** L: Edge_Label;
rpl ED: Edge_Data_Rec);
requires 1 ≤ V1 ≤ Vertex_Max **and** 1 ≤ V2 ≤ Vertex_Max **and** C > 0.0;
ensures ED.Vert1 = V1 **and** ED.Vert2 = V2 **and** ED.Cst = C **and** ED.Lab = @L;

Oper Recover_Data(**rpl** V1, V2: Integer; **rpl** C: Real; **rpl** L: Edge_Label;
alt ED: Edge_Data_Rec);
ensures V1 = @ED.Vert1 **and** V2 = @ED.Vert2 **and** C = @ED.Cst **and**
L = @ED.Lab;

Def const (ED1: Edge_Data_Rec) $\preceq_{Cst}$ (ED2: Edge_Data_Rec): $\mathbb{B}$ =
(ED1.Cst ≤ ED2.Cst);

Oper Is_Cheaper_Edge(**rest** ED1, ED2: Edge_Data_Rec): Boolean;
ensures Is_Cheaper_Edge = (ED1 $\precsim_{Cst}$ ED2);
procedure
Is_Cheaper_Edge := (ED1.Cst ≤ ED2.Cst);
end Is_Cheaper_Edge;
end Edge_Record_Fac;

Def. Edge_Universe = **Cart_Prod**
Inf: Edge_Data_Rec;
Inst_Num: [1..Max_Edge_Count]
end;

Facility Edge_Priorit_Fac **is** Prioritizer_Template(Edge_Data_Rec, Max_Edge_Count, $\precsim_{Cst}$)
realized_by Arb_Pri_Realiz(Is_Cheaper_Edge);

Facility Connect_Reln_Fac **is** Coalescable_Equivalence_Relation_Template(Vertex_Max)
realized_by Arb_Equiv_Realiz;

Type Graph_Holder = **Record**
Edge_Keeper: Edge_Priorit_Fac.Entry_Keeper;
Are_Connected_Reln: Connect_Reln_Fac.Equiv_Reln;
Span_Edge_Count: Integer
end;
Implicit Def const
Graph_from(EK: Edge_Priorit_Fac.Entry_Keeper): **conc**.Graph_Holder **is**
Graph_from(EK).Span_Found = ¬ EK.Accepting **and** ∀ E: Edge_Universe,
Graph_from(EK).Is_Gr_Edge(E) = (E.Inst_Num ≤ EK.Entry_Count(E.Inf)) **and**
Graph_from(EK).E_Info(E).Ends = {E.Inf.Vert1, E.Inf.Vert2} **and**
Graph_from(EK).E_Info(E).Cost = E.Inf.Cst **and**
Graph_from(EK).E_Info(E).Label = E.Inf.Lab;

Implicit Def const Prune_by(CR: Connect_Reln_Fac.Equiv_Reln;
EK: Edge_Priorit_Fac.Entry_Keeper): Edge_Priorit_Fac.Entry_Keeper **is**
Prune_by(CR, EK).Accepting = EK.Accepting **and** ∀ ED: Edge_Data_Rec,
Prune_by(CR, EK).Entry_Count(ED) =

$$\begin{cases} 0 & \text{if } CR(ED.Vert1, ED.Vert2) \\ EK.Entry\_Count(ED) & \text{otherwise} \end{cases};$$

conventions
if GH.Edge_Keeper.Accepting **then**
∀ V1, V2: Vertex, GH.Are_Connected_Reln(V1, V2) =
Are_Connected_In(Graph_from(GH.Edge_Keeper), V1, V2) **and**
∀ MGH: **conc**.Graph_Holder,
if (MGH)Is_Min_Span(Graph_from(GH.Edge_Keeper))
then Edge_Count(MGH) = GH.Span_Edge_Count **and**
if ¬ GH.Edge_Keeper.Accepting **then** ∀ MGH: **conc**.Graph_Holder,

```
            if  ( MGH )Is_Min_Span( Graph_from(
                                    Prune_by(GH.Are_Connected_Reln, GH.Edge_Keeper) ) )
                then Edge_Count(MGH) = GH.Span_Edge_Count;
        correspondence
            if GH.Edge_Keeper.Accepting then conc.GH = Graph_from( GH.Edge_Keeper )
            and if ¬ GH.Edge_Keeper.Accepting then
                ( conc.GH )Is_Min_Span( Graph_from(
                                    Prune_by(GH.Are_Connected_Reln, GH.Edge_Keeper) ) );

    Proc Augment_w( eval V1, V2: Integer; eval C: Real; alt L: Edge_Label;
                    upd GH: Graph_Holder, aux AGH: Graph_Holder, aux NE: Edge_Universe );
            ⋮
end Kruskal_Realiz;
```