

LinksPlatform's Platform.Data.Triplets Class Library

1.1 ./csharp/Platform.Data.Triplets/CharacterHelpers.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Triplets
7  {
8      // TODO: Split logic of Latin and Cyrillic alphabets into different files if possible
9      public static class CharacterHelpers
10     {
11         public enum CharacterMapping : long
12         {
13             LatinAlphabet = 100,
14             CyrillicAlphabet
15         }
16
17         private const char FirstLowerCaseLatinLetter = 'a';
18         private const char LastLowerCaseLatinLetter = 'z';
19         private const char FirstUpperCaseLatinLetter = 'A';
20         private const char LastUpperCaseLatinLetter = 'Z';
21         private const char FirstLowerCaseCyrillicLetter = 'а';
22         private const char LastLowerCaseCyrillicLetter = 'я';
23         private const char FirstUpperCaseCyrillicLetter = 'А';
24         private const char LastUpperCaseCyrillicLetter = 'Я';
25         private const char YoLowerCaseCyrillicLetter = 'ё';
26         private const char YoUpperCaseCyrillicLetter = 'Ё';
27
28         private static Link[] _charactersToLinks;
29         private static Dictionary<Link, char> _linksToCharacters;
30
31         static CharacterHelpers() => Create();
32
33         private static void Create()
34         {
35             _charactersToLinks = new Link[char.MaxValue];
36             _linksToCharacters = new Dictionary<Link, char>();
37             // Create or restore characters
38             CreateLatinAlphabet();
39             CreateCyrillicAlphabet();
40             RegisterExistingCharacters();
41         }
42
43         private static void RegisterExistingCharacters() =>
44         ↪ Net.Character.WalkThroughReferersAsSource(referer =>
45         ↪ RegisterExistingCharacter(referer));
46
47         private static void RegisterExistingCharacter(Link character)
48         {
49             if (character.Source == Net.Character && character.Linker == Net.ThatHas)
50             {
51                 var code = character.Target;
52                 if (code.Source == Net.Code && code.Linker == Net.ThatIsRepresentedBy)
53                 {
54                     var charCode = (char)LinkConverter.ToNumber(code.Target);
55                     _charactersToLinks[charCode] = character;
56                     _linksToCharacters[character] = charCode;
57                 }
58             }
59         }
60
61         public static void Recreate() => Create();
62
63         private static void CreateLatinAlphabet()
64         {
65             var lettersCharacters = new[]
66             {
67                 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
68                 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
69                 'u', 'v', 'w', 'x', 'y', 'z'
70             };
71             CreateAlphabet(lettersCharacters, "latin alphabet", CharacterMapping.LatinAlphabet);
72         }
73
74         private static void CreateCyrillicAlphabet()
75         {
76             var lettersCharacters = new[]
77             {
78                 'а', 'б', 'в', 'г', 'д', 'е', 'ё', 'ж', 'з', 'и',
```

```

77         'й', 'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с', 'т',
78         'у', 'ф', 'х', 'ц', 'ч', 'ш', 'щ', 'ъ', 'ы', 'ь',
79         'э', 'ю', 'я'
80     };
81     CreateAlphabet(lettersCharacters, "cyrillic alphabet",
82         ↪ CharacterMapping.CyrillicAlphabet);
83 }
84 private static void CreateAlphabet(char[] lettersCharacters, string alphabetName,
85     ↪ CharacterMapping mapping)
86 {
87     if (Link.TryGetMapped(mapping, out Link alphabet))
88     {
89         var letters = alphabet.Target;
90         letters.WalkThroughSequence(letter =>
91         {
92             var lowerCaseLetter = Link.Search(Net.LowerCase, Net.Of, letter);
93             var upperCaseLetter = Link.Search(Net.UpperCase, Net.Of, letter);
94             if (lowerCaseLetter != null && upperCaseLetter != null)
95             {
96                 RegisterExistingLetter(lowerCaseLetter);
97                 RegisterExistingLetter(upperCaseLetter);
98             }
99             else
100             {
101                 RegisterExistingLetter(letter);
102             }
103         });
104     }
105     else
106     {
107         alphabet = Net.CreateMappedThing(mapping);
108         var letterOfAlphabet = Link.Create(Net.Letter, Net.Of, alphabet);
109         var lettersLinks = new Link[lettersCharacters.Length];
110         GenerateAlphabetBasis(ref alphabet, ref letterOfAlphabet, lettersLinks);
111         for (var i = 0; i < lettersCharacters.Length; i++)
112         {
113             var lowerCaseCharacter = lettersCharacters[i];
114             SetLetterCodes(lettersLinks[i], lowerCaseCharacter, out Link lowerCaseLink,
115                 ↪ out Link upperCaseLink);
116             _charactersToLinks[lowerCaseCharacter] = lowerCaseLink;
117             _linksToCharacters[lowerCaseLink] = lowerCaseCharacter;
118             if (upperCaseLink != null)
119             {
120                 var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
121                 _charactersToLinks[upperCaseCharacter] = upperCaseLink;
122                 _linksToCharacters[upperCaseLink] = upperCaseCharacter;
123             }
124         }
125         alphabet.SetName(alphabetName);
126         for (var i = 0; i < lettersCharacters.Length; i++)
127         {
128             var lowerCaseCharacter = lettersCharacters[i];
129             var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
130             if (lowerCaseCharacter != upperCaseCharacter)
131             {
132                 lettersLinks[i].SetName("{ " + upperCaseCharacter + " " +
133                     ↪ lowerCaseCharacter + "}");
134             }
135             else
136             {
137                 lettersLinks[i].SetName("{ " + lowerCaseCharacter + "}");
138             }
139         }
140     }
141 }
142 private static void RegisterExistingLetter(Link letter)
143 {
144     letter.WalkThroughReferersAsSource(referer =>
145     {
146         if (referer.Linker == Net.Has)
147         {
148             var target = referer.Target;
149             if (target.Source == Net.Code && target.Linker ==
150                 ↪ Net.ThatIsRepresentedBy)
151             {
152                 var charCode = (char)LinkConverter.ToNumber(target.Target);

```

```

150         _charactersToLinks[charCode] = letter;
151         _linksToCharacters[letter] = charCode;
152     }
153 }
154 });
155 }
156
157 private static void GenerateAlphabetBasis(ref Link alphabet, ref Link letterOfAlphabet,
158     ↪ Link[] letters)
159 {
160     // Принцип, на примере латинского алфавита.
161     // latin alphabet: alphabet that consists of a and b and c and ... and z.
162     // a: letter of latin alphabet that is before b.
163     // b: letter of latin alphabet that is between (a and c).
164     // c: letter of latin alphabet that is between (b and e).
165     // ...
166     // y: letter of latin alphabet that is between (x and z).
167     // z: letter of latin alphabet that is after y.
168     const int firstLetterIndex = 0;
169     for (var i = firstLetterIndex; i < letters.Length; i++)
170     {
171         letters[i] = Net.CreateThing();
172     }
173     var lastLetterIndex = letters.Length - 1;
174     Link.Update(ref letters[firstLetterIndex], letterOfAlphabet, Net.ThatIsBefore,
175         ↪ letters[firstLetterIndex + 1]);
176     Link.Update(ref letters[lastLetterIndex], letterOfAlphabet, Net.ThatIsAfter,
177         ↪ letters[lastLetterIndex - 1]);
178     const int secondLetterIndex = firstLetterIndex + 1;
179     for (var i = secondLetterIndex; i < lastLetterIndex; i++)
180     {
181         Link.Update(ref letters[i], letterOfAlphabet, Net.ThatIsBetween, letters[i - 1]
182             ↪ & letters[i + 1]);
183     }
184     Link.Update(ref alphabet, Net.Alphabet, Net.ThatConsistsOf,
185         ↪ LinkConverter.FromList(letters));
186 }
187
188 private static void SetLetterCodes(Link letter, char lowerCaseCharacter, out Link
189     ↪ lowerCase, out Link upperCase)
190 {
191     var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
192     if (upperCaseCharacter != lowerCaseCharacter)
193     {
194         lowerCase = Link.Create(Net.LowerCase, Net.Of, letter);
195         var lowerCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
196             ↪ LinkConverter.FromNumber(lowerCaseCharacter));
197         Link.Create(lowerCase, Net.Has, lowerCaseCharacterCode);
198         upperCase = Link.Create(Net.UpperCase, Net.Of, letter);
199         var upperCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
200             ↪ LinkConverter.FromNumber(upperCaseCharacter));
201         Link.Create(upperCase, Net.Has, upperCaseCharacterCode);
202     }
203     else
204     {
205         lowerCase = letter;
206         upperCase = null;
207         Link.Create(letter, Net.Has, Link.Create(Net.Code, Net.ThatIsRepresentedBy,
208             ↪ LinkConverter.FromNumber(lowerCaseCharacter)));
209     }
210 }
211
212 private static Link CreateSimpleCharacterLink(char character) =>
213     ↪ Link.Create(Net.Character, Net.ThatHas, Link.Create(Net.Code,
214     ↪ Net.ThatIsRepresentedBy, LinkConverter.FromNumber(character)));
215
216 private static bool IsLetterOfLatinAlphabet(char character)
217     => (character >= FirstLowerCaseLatinLetter && character <= LastLowerCaseLatinLetter)
218     || (character >= FirstUpperCaseLatinLetter && character <= LastUpperCaseLatinLetter);
219
220 private static bool IsLetterOfCyrillicAlphabet(char character)
221     => (character >= FirstLowerCaseCyrillicLetter && character <=
222     ↪ LastLowerCaseCyrillicLetter)
223     || (character >= FirstUpperCaseCyrillicLetter && character <=
224     ↪ LastUpperCaseCyrillicLetter)
225     || character == YoLowerCaseCyrillicLetter || character == YoUpperCaseCyrillicLetter;
226
227 public static Link FromChar(char character)

```

```

215 {
216     if (_charactersToLinks[character] == null)
217     {
218         if (IsLetterOfLatinAlphabet(character))
219         {
220             CreateLatinAlphabet();
221             return _charactersToLinks[character];
222         }
223         else if (IsLetterOfCyrillicAlphabet(character))
224         {
225             CreateCyrillicAlphabet();
226             return _charactersToLinks[character];
227         }
228         else
229         {
230             var simpleCharacter = CreateSimpleCharacterLink(character);
231             _charactersToLinks[character] = simpleCharacter;
232             _linksToCharacters[simpleCharacter] = character;
233             return simpleCharacter;
234         }
235     }
236     else
237     {
238         return _charactersToLinks[character];
239     }
240 }
241
242 public static char ToChar(Link link)
243 {
244     if (!_linksToCharacters.TryGetValue(link, out char @char))
245     {
246         throw new ArgumentOutOfRangeException(nameof(link), "Указанная связь не
247             ↳ является символом.");
248     }
249     return @char;
250 }
251
252 public static bool IsChar(Link link) => link != null &&
253     ↳ _linksToCharacters.ContainsKey(link);
254 }
255 }

```

1.2 ./csharp/Platform.Data.Triplets/GexfExporter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.IO;
4  using System.Text;
5  using System.Xml;
6  using Platform.Collections.Sets;
7  using Platform.Communication.Protocol.Gexf;
8  using GexfNode = Platform.Communication.Protocol.Gexf.Node;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Triplets
13 {
14     public static class GexfExporter
15     {
16         private const string SourceLabel = "Source";
17         private const string LinkerLabel = "Linker";
18         private const string TargetLabel = "Target";
19
20         public static string ToXml()
21         {
22             var sb = new StringBuilder();
23             using (var writer = XmlWriter.Create(sb))
24             {
25                 WriteXml(writer, CollectLinks());
26             }
27             return sb.ToString();
28         }
29
30         public static void ToFile(string path)
31         {
32             using (var file = File.OpenWrite(path))
33             using (var writer = XmlWriter.Create(file))
34             {
35                 WriteXml(writer, CollectLinks());
36             }
37         }
38     }
39 }

```

```

37     }
38
39     public static void ToFile(string path, Func<Link, bool> filter)
40     {
41         using (var file = File.OpenWrite(path))
42         using (var writer = XmlWriter.Create(file))
43         {
44             WriteXml(writer, CollectLinks(filter));
45         }
46     }
47
48     private static HashSet<Link> CollectLinks(Func<Link, bool> linkMatch)
49     {
50         var matchingLinks = new HashSet<Link>();
51         Link.WalkThroughAllLinks(link =>
52         {
53             if (linkMatch(link))
54             {
55                 matchingLinks.Add(link);
56             }
57         });
58         return matchingLinks;
59     }
60
61     private static HashSet<Link> CollectLinks()
62     {
63         var matchingLinks = new HashSet<Link>();
64         Link.WalkThroughAllLinks(matchingLinks.AddAndReturnVoid);
65         return matchingLinks;
66     }
67
68     private static void WriteXml(XmlWriter writer, HashSet<Link> matchingLinks)
69     {
70         var edgesCounter = 0;
71         Gexf.WriteXml(writer,
72             () => // nodes
73             {
74                 foreach (var matchingLink in matchingLinks)
75                 {
76                     GexfNode.WriteXml(writer, matchingLink.ToInt(), matchingLink.ToString());
77                 }
78             },
79             () => // edges
80             {
81                 foreach (var matchingLink in matchingLinks)
82                 {
83                     if (matchingLinks.Contains(matchingLink.Source))
84                     {
85                         Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
86                             ↪ matchingLink.Source.ToInt(), SourceLabel);
87                     }
88                     if (matchingLinks.Contains(matchingLink.Linker))
89                     {
90                         Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
91                             ↪ matchingLink.Linker.ToInt(), LinkerLabel);
92                     }
93                     if (matchingLinks.Contains(matchingLink.Target))
94                     {
95                         Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
96                             ↪ matchingLink.Target.ToInt(), TargetLabel);
97                     }
98                 }
99             });
100     }

```

1.3 ./csharp/Platform.Data.Triplets/ILink.cs

```

1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Triplets
6  {
7      internal partial interface ILink<TLink>
8          where TLink : ILink<TLink>
9      {
10         TLink Source { get; }

```

```

11     TLink Linker { get; }
12     TLink Target { get; }
13 }
14
15 internal partial interface ILink<TLink>
16     where TLink : ILink<TLink>
17 {
18     bool WalkThroughReferersAsLinker(Func<TLink, bool> walker);
19     bool WalkThroughReferersAsSource(Func<TLink, bool> walker);
20     bool WalkThroughReferersAsTarget(Func<TLink, bool> walker);
21     void WalkThroughReferers(Func<TLink, bool> walker);
22 }
23
24 internal partial interface ILink<TLink>
25     where TLink : ILink<TLink>
26 {
27     void WalkThroughReferersAsLinker(Action<TLink> walker);
28     void WalkThroughReferersAsSource(Action<TLink> walker);
29     void WalkThroughReferersAsTarget(Action<TLink> walker);
30     void WalkThroughReferers(Action<TLink> walker);
31 }
32 }
33 /*
34 using System;
35 namespace NetLibrary
36 {
37     interface ILink
38     {
39         // Статические методы (общие для всех связей)
40         public static ILink Create(ILink source, ILink linker, ILink target);
41         public static void Update(ref ILink link, ILink newSource, ILink newLinker, ILink
↵ newTarget);
42         public static void Delete(ref ILink link);
43         public static ILink Search(ILink source, ILink linker, ILink target);
44     }
45 }
46 */
47 /*
48 Набор функций, который необходим для работы с сущностью Link:
49
50 (Работа со значением сущности Link, значение состоит из 3-х частей, также сущностей Link)
51 1. Получить адрес "начальной" сущности Link. (Получить адрес из поля Source)
52 2. Получить адрес сущности Link, которая играет роль связки между "начальной" и "конечной"
↵ сущностями Link. (Получить адрес из поля Linker)
53 3. Получить адрес "конечной" сущности Link. (Получить адрес из поля Target)
54
55 4. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↵ которых поле Source равно этому адресу.
56 5. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↵ которых поле Linker равно этому адресу.
57 6. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↵ которых поле Target равно этому адресу.
58
59 7. Создать сущность Link со значением (смысл), которым являются адреса на другие 3 сущности
↵ Link (где первая является "начальной", вторая является "связкой", а третья является
↵ "конечной").
60 8. Обновление сущности Link с указанным адресом новым значением (смысл), которым являются
↵ адреса на другие 3 сущности Link (где первая является "начальной", вторая является
↵ "связкой", а третья является "конечной").
61 9. Удаление сущности Link с указанным адресом.
62 10. Поиск сущности Link со значением (смысл), которым являются адреса на другие 3 сущности
↵ Link (где первая является "начальной", вторая является "связкой", а третья является
↵ "конечной").
63 */

```

1.4 ./csharp/Platform.Data.Triplets/Link.Debug.cs

```

1 using System;
2 using System.Diagnostics;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Triplets
7 {
8     public partial struct Link
9     {
10         #region Properties
11
12         // ReSharper disable InconsistentNaming
13         // ReSharper disable UnusedMember.Local

```

```

14 #pragma warning disable IDE0051 // Remove unused private members
15
16 [DebuggerDisplay(null, Name = "Source")]
17 private Link Я_A => this == null ? Itself : Source;
18
19 [DebuggerDisplay(null, Name = "Linker")]
20 private Link Я_B => this == null ? Itself : Linker;
21
22 [DebuggerDisplay(null, Name = "Target")]
23 private Link Я_C => this == null ? Itself : Target;
24
25 [DebuggerDisplay("Count = {Я_DC}", Name = "ReferersBySource")]
26 private Link[] Я_D => this.GetArrayOfRererersBySource();
27
28 [DebuggerDisplay("Count = {Я_EC}", Name = "ReferersByLinker")]
29 private Link[] Я_E => this.GetArrayOfRererersByLinker();
30
31 [DebuggerDisplay("Count = {Я_FC}", Name = "ReferersByTarget")]
32 private Link[] Я_F => this.GetArrayOfRererersByTarget();
33
34 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
35 private Int64 Я_DC => this == null ? 0 : ReferersBySourceCount;
36
37 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
38 private Int64 Я_EC => this == null ? 0 : ReferersByLinkerCount;
39
40 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
41 private Int64 Я_FC => this == null ? 0 : ReferersByTargetCount;
42
43 [DebuggerDisplay(null, Name = "Timestamp")]
44 private DateTime Я_H => this == null ? DateTime.MinValue : Timestamp;
45
46 // ReSharper restore UnusedMember.Local
47 // ReSharper restore InconsistentNaming
48 #pragma warning restore IDE0051 // Remove unused private members
49
50 #endregion
51
52 public override string ToString()
53 {
54     const string nullString = "null";
55     if (this == null)
56     {
57         return nullString;
58     }
59     else
60     {
61         if (this.TryGetName(out string name))
62         {
63             return name;
64         }
65         else
66         {
67             return ((long)_link).ToString();
68         }
69     }
70 }
71 }
72 }

```

1.5 ./csharp/Platform.Data.Triplets/Link.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Runtime.InteropServices;
5 using System.Threading;
6 using Int = System.Int64;
7 using LinkIndex = System.UInt64;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Triplets
12 {
13     public struct LinkDefinition : IEquatable<LinkDefinition>
14     {
15         public readonly Link Source;
16         public readonly Link Linker;
17         public readonly Link Target;
18
19         public LinkDefinition(Link source, Link linker, Link target)
20         {

```

```

21         Source = source;
22         Linker = linker;
23         Target = target;
24     }
25
26     public LinkDefinition(Link link) : this(link.Source, link.Linker, link.Target) { }
27
28     public bool Equals(LinkDefinition other) => Source == other.Source && Linker ==
        ↳ other.Linker && Target == other.Target;
29 }
30
31 public partial struct Link : ILink<Link>, IEquatable<Link>
32 {
33     private const string DllName = "Platform_Data_Triplets_Kernel";
34
35     // TODO: Заменить на очередь событий, по примеру Node.js (+сделать выключаемым)
36     public delegate void CreatedDelegate(LinkDefinition createdLink);
37     public static event CreatedDelegate CreatedEvent = createdLink => { };
38
39     public delegate void UpdatedDelegate(LinkDefinition linkBeforeUpdate, LinkDefinition
        ↳ linkAfterUpdate);
40     public static event UpdatedDelegate UpdatedEvent = (linkBeforeUpdate, linkAfterUpdate)
        ↳ => { };
41
42     public delegate void DeletedDelegate(LinkDefinition deletedLink);
43     public static event DeletedDelegate DeletedEvent = deletedLink => { };
44
45     #region Low Level
46
47     #region Basic Operations
48
49     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
50     private static extern LinkIndex GetSourceIndex(LinkIndex link);
51
52     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
53     private static extern LinkIndex GetLinkerIndex(LinkIndex link);
54
55     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
56     private static extern LinkIndex GetTargetIndex(LinkIndex link);
57
58     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
59     private static extern LinkIndex GetFirstRefererBySourceIndex(LinkIndex link);
60
61     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
62     private static extern LinkIndex GetFirstRefererByLinkerIndex(LinkIndex link);
63
64     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
65     private static extern LinkIndex GetFirstRefererByTargetIndex(LinkIndex link);
66
67     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
68     private static extern Int GetTime(LinkIndex link);
69
70     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
71     private static extern LinkIndex CreateLink(LinkIndex source, LinkIndex linker, LinkIndex
        ↳ target);
72
73     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
74     private static extern LinkIndex UpdateLink(LinkIndex link, LinkIndex newSource,
        ↳ LinkIndex newLinker, LinkIndex newTarget);
75
76     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
77     private static extern void DeleteLink(LinkIndex link);
78
79     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
80     private static extern LinkIndex ReplaceLink(LinkIndex link, LinkIndex replacement);
81
82     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
83     private static extern LinkIndex SearchLink(LinkIndex source, LinkIndex linker, LinkIndex
        ↳ target);
84
85     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
86     private static extern LinkIndex GetMappedLink(Int mappedIndex);
87
88     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
89     private static extern void SetMappedLink(Int mappedIndex, LinkIndex linkIndex);
90
91     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
92     private static extern Int OpenLinks(string filename);
93
94     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]

```



```

95     private static extern Int CloseLinks();
96
97     #endregion
98
99     #region Referers Count Selectors
100
101     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
102     private static extern LinkIndex GetLinkNumberOfReferersBySource(LinkIndex link);
103
104     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
105     private static extern LinkIndex GetLinkNumberOfReferersByLinker(LinkIndex link);
106
107     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
108     private static extern LinkIndex GetLinkNumberOfReferersByTarget(LinkIndex link);
109
110     #endregion
111
112     #region Referers Walkers
113
114     private delegate void Visitor(LinkIndex link);
115     private delegate Int StopableVisitor(LinkIndex link);
116
117     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
118     private static extern void WalkThroughAllReferersBySource(LinkIndex root, Visitor
        ↪ action);
119
120     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
121     private static extern int WalkThroughReferersBySource(LinkIndex root, StopableVisitor
        ↪ func);
122
123     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
124     private static extern void WalkThroughAllReferersByLinker(LinkIndex root, Visitor
        ↪ action);
125
126     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
127     private static extern int WalkThroughReferersByLinker(LinkIndex root, StopableVisitor
        ↪ func);
128
129     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
130     private static extern void WalkThroughAllReferersByTarget(LinkIndex root, Visitor
        ↪ action);
131
132     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
133     private static extern int WalkThroughReferersByTarget(LinkIndex root, StopableVisitor
        ↪ func);
134
135     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
136     private static extern void WalkThroughAllLinks(Visitor action);
137
138     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
139     private static extern int WalkThroughLinks(StopableVisitor func);
140
141     #endregion
142
143     #endregion
144
145     #region Constains
146
147     public static readonly Link Itself = null;
148     public static readonly bool Continue = true;
149     public static readonly bool Stop;
150
151     #endregion
152
153     #region Static Fields
154
155     private static readonly object _lockObject = new object();
156     private static volatile int _memoryManagerIsReady;
157     private static readonly Dictionary<ulong, long> _linkToMappingIndex = new
        ↪ Dictionary<ulong, long>();
158
159     #endregion
160
161     #region Fields
162
163     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
164     private readonly LinkIndex _link;
165
166     #endregion
167
168     #region Properties

```

```

169 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
170 public Link Source => GetSourceIndex(_link);
171
172 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
173 public Link Linker => GetLinkerIndex(_link);
174
175 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
176 public Link Target => GetTargetIndex(_link);
177
178 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
179 public Link FirstRefererBySource => GetFirstRefererBySourceIndex(_link);
180
181 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
182 public Link FirstRefererByLinker => GetFirstRefererByLinkerIndex(_link);
183
184 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
185 public Link FirstRefererByTarget => GetFirstRefererByTargetIndex(_link);
186
187 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
188 public Int ReferersBySourceCount => (Int)GetLinkNumberOfReferersBySource(_link);
189
190 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
191 public Int ReferersByLinkerCount => (Int)GetLinkNumberOfReferersByLinker(_link);
192
193 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
194 public Int ReferersByTargetCount => (Int)GetLinkNumberOfReferersByTarget(_link);
195
196 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
197 public Int TotalReferers => (Int)GetLinkNumberOfReferersBySource(_link) +
198     ↳ (Int)GetLinkNumberOfReferersByLinker(_link) +
199     ↳ (Int)GetLinkNumberOfReferersByTarget(_link);
200
201 [DebuggerBrowsable(DebuggerBrowsableState.Never)]
202 public DateTime Timestamp => DateTime.FromFileTimeUtc(GetTime(_link));
203
204 #endregion
205
206 #region Infrastructure
207
208 public Link(LinkIndex link) => _link = link;
209
210 public static void StartMemoryManager(string storageFilename)
211 {
212     lock (_lockObject)
213     {
214         if (_memoryManagerIsReady == default)
215         {
216             if (OpenLinks(storageFilename) == 0)
217             {
218                 throw new InvalidOperationException($"Файл ({storageFilename})
219                     ↳ хранилища не удалось открыть.");
220             }
221             Interlocked.Exchange(ref _memoryManagerIsReady, 1);
222         }
223     }
224
225 public static void StopMemoryManager()
226 {
227     lock (_lockObject)
228     {
229         if (_memoryManagerIsReady != default)
230         {
231             if (CloseLinks() == 0)
232             {
233                 throw new InvalidOperationException("Файл хранилища не удалось закрыть.
234                     ↳ Возможно он был уже закрыт, или не открывался вовсе.");
235             }
236             Interlocked.Exchange(ref _memoryManagerIsReady, 0);
237         }
238     }
239
240 public static implicit operator LinkIndex?(Link link) => link._link == 0 ?
241     ↳ (LinkIndex?)null : link._link;
242
243 public static implicit operator Link(LinkIndex? link) => new Link(link ?? 0);

```

```

243 public static implicit operator Int(Link link) => (Int)link._link;
244
245 public static implicit operator Link(Int link) => new Link((LinkIndex)link);
246
247 public static implicit operator LinkIndex(Link link) => link._link;
248
249 public static implicit operator Link(LinkIndex link) => new Link(link);
250
251 public static explicit operator Link(List<Link> links) => LinkConverter.FromList(links);
252
253 public static explicit operator Link(Link[] links) => LinkConverter.FromList(links);
254
255 public static explicit operator Link(string @string) =>
    ↳ LinkConverter.FromString(@string);
256
257 public static bool operator ==(Link first, Link second) => first.Equals(second);
258
259 public static bool operator !=(Link first, Link second) => !first.Equals(second);
260
261 public static Link operator &(Link first, Link second) => Create(first, Net.And, second);
262
263 public override bool Equals(object obj) => obj is Link link ? Equals(link) : false;
264
265 public bool Equals(Link other) => _link == other._link || (LinkDoesNotExist(_link) &&
    ↳ LinkDoesNotExist(other._link));
266
267 public override int GetHashCode() => base.GetHashCode();
268
269 private static bool LinkDoesNotExist(LinkIndex link) => link == 0 ||
    ↳ GetLinkerIndex(link) == 0;
270
271 private static bool LinkWasDeleted(LinkIndex link) => link != 0 && GetLinkerIndex(link)
    ↳ == 0;
272
273 private bool IsMatchingTo(Link source, Link linker, Link target)
274     => ((Source == this && source == null) || (Source == source))
275     && ((Linker == this && linker == null) || (Linker == linker))
276     && ((Target == this && target == null) || (Target == target));
277
278 public LinkIndex ToIndex() => _link;
279
280 public Int ToInt() => (Int)_link;
281
282 #endregion
283
284 #region Basic Operations
285
286 public static Link Create(Link source, Link linker, Link target)
287 {
288     if (_memoryManagerIsReady == default)
289     {
290         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
291     }
292     if (LinkWasDeleted(source))
293     {
294         throw new ArgumentException("Удалённая связь не может использоваться в качестве
            ↳ значения.", nameof(source));
295     }
296     if (LinkWasDeleted(linker))
297     {
298         throw new ArgumentException("Удалённая связь не может использоваться в качестве
            ↳ значения.", nameof(linker));
299     }
300     if (LinkWasDeleted(target))
301     {
302         throw new ArgumentException("Удалённая связь не может использоваться в качестве
            ↳ значения.", nameof(target));
303     }
304     Link link = CreateLink(source, linker, target);
305     if (link == null)
306     {
307         throw new InvalidOperationException("Невозможно создать связь.");
308     }
309     CreatedEvent.Invoke(new LinkDefinition(link));
310     return link;
311 }
312
313 public static Link Restore(Int index) => Restore((LinkIndex)index);
314

```

```

315 public static Link Restore(LinkIndex index)
316 {
317     if (_memoryManagerIsReady == default)
318     {
319         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
320     }
321     if (index == 0)
322     {
323         throw new ArgumentException("У связи не может быть нулевого адреса.");
324     }
325     try
326     {
327         Link link = index;
328         if (LinkDoesNotExist(link))
329         {
330             throw new InvalidOperationException("Связь с указанным адресом удалена, либо
↪ не существовала.");
331         }
332         return link;
333     }
334     catch (Exception ex)
335     {
336         throw new InvalidOperationException("Указатель не является корректным.", ex);
337     }
338 }
339
340 public static Link CreateMapped(Link source, Link linker, Link target, object
↪ mappingIndex) => CreateMapped(source, linker, target, Convert.ToInt64(mappingIndex));
341
342 public static Link CreateMapped(Link source, Link linker, Link target, Int mappingIndex)
343 {
344     if (_memoryManagerIsReady == default)
345     {
346         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
347     }
348     Link mappedLink = GetMappedLink(mappingIndex);
349     if (mappedLink == null)
350     {
351         mappedLink = Create(source, linker, target);
352         SetMappedLink(mappingIndex, mappedLink);
353         if (GetMappedLink(mappingIndex) != mappedLink)
354         {
355             throw new InvalidOperationException("Установить привязанную связь не
↪ удалось.");
356         }
357     }
358     else if (!mappedLink.IsMatchingTo(source, linker, target))
359     {
360         throw new InvalidOperationException("Существующая привязанная связь не
↪ соответствует указанным Source, Linker и Target.");
361     }
362     _linkToMappingIndex[mappedLink] = mappingIndex;
363     return mappedLink;
364 }
365
366 public static bool TrySetMapped(Link link, Int mappingIndex, bool rewrite = false)
367 {
368     Link mappedLink = GetMappedLink(mappingIndex);
369
370     if (mappedLink == null || rewrite)
371     {
372         mappedLink = link;
373         SetMappedLink(mappingIndex, mappedLink);
374         if (GetMappedLink(mappingIndex) != mappedLink)
375         {
376             return false;
377         }
378     }
379     else if (!mappedLink.IsMatchingTo(link.Source, link.Linker, link.Target))
380     {
381         return false;
382     }
383     _linkToMappingIndex[mappedLink] = mappingIndex;
384     return true;
385 }
386
387 public static Link GetMapped(object mappingIndex) =>
↪ GetMapped(Convert.ToInt64(mappingIndex));

```

```

388 public static Link GetMapped(Int mappingIndex)
389 {
390     if (!TryGetMapped(mappingIndex, out Link mappedLink))
391     {
392         throw new InvalidOperationException($"Mapped link with index {mappingIndex} is
393             ↪ not set.");
394     }
395     return mappedLink;
396 }
397
398 public static Link GetMappedOrDefault(object mappingIndex)
399 {
400     TryGetMapped(mappingIndex, out Link mappedLink);
401     return mappedLink;
402 }
403
404 public static Link GetMappedOrDefault(Int mappingIndex)
405 {
406     TryGetMapped(mappingIndex, out Link mappedLink);
407     return mappedLink;
408 }
409
410 public static bool TryGetMapped(object mappingIndex, out Link mappedLink) =>
411     ↪ TryGetMapped(Convert.ToInt64(mappingIndex), out mappedLink);
412
413 public static bool TryGetMapped(Int mappingIndex, out Link mappedLink)
414 {
415     if (_memoryManagerIsReady == default)
416     {
417         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
418     }
419     mappedLink = GetMappedLink(mappingIndex);
420     if (mappedLink != null)
421     {
422         _linkToMappingIndex[mappedLink] = mappingIndex;
423     }
424     return mappedLink != null;
425 }
426
427 public static Link Update(Link link, Link newSource, Link newLinker, Link newTarget)
428 {
429     Update(ref link, newSource, newLinker, newTarget);
430     return link;
431 }
432
433 public static void Update(ref Link link, Link newSource, Link newLinker, Link newTarget)
434 {
435     if (_memoryManagerIsReady == default)
436     {
437         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
438     }
439     if (LinkDoesNotExist(link))
440     {
441         throw new ArgumentException("Нельзя обновить несуществующую связь.",
442             ↪ nameof(link));
443     }
444     if (LinkWasDeleted(newSource))
445     {
446         throw new ArgumentException("Удалённая связь не может использоваться в качестве
447             ↪ нового значения.", nameof(newSource));
448     }
449     if (LinkWasDeleted(newLinker))
450     {
451         throw new ArgumentException("Удалённая связь не может использоваться в качестве
452             ↪ нового значения.", nameof(newLinker));
453     }
454     if (LinkWasDeleted(newTarget))
455     {
456         throw new ArgumentException("Удалённая связь не может использоваться в качестве
457             ↪ нового значения.", nameof(newTarget));
458     }
459     LinkIndex previousLinkIndex = link;
460     _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
461     var previousDefinition = new LinkDefinition(link);
462     link = UpdateLink(link, newSource, newLinker, newTarget);
463     if (mappingIndex >= 0 && previousLinkIndex != link)
464     {

```

```

460         _linkToMappingIndex.Remove(previousLinkIndex);
461         SetMappedLink(mappingIndex, link);
462         _linkToMappingIndex.Add(link, mappingIndex);
463     }
464     UpdatedEvent(previousDefinition, new LinkDefinition(link));
465 }
466
467 public static void Delete(Link link) => Delete(ref link);
468
469 public static void Delete(ref Link link)
470 {
471     if (LinkDoesNotExist(link))
472     {
473         return;
474     }
475     LinkIndex previousLinkIndex = link;
476     _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
477     var previousDefinition = new LinkDefinition(link);
478     DeleteLink(link);
479     link = null;
480     if (mappingIndex >= 0)
481     {
482         _linkToMappingIndex.Remove(previousLinkIndex);
483         SetMappedLink(mappingIndex, 0);
484     }
485     DeletedEvent(previousDefinition);
486 }
487
488 //public static void Replace(ref Link link, Link replacement)
489 //{
490 //    if (!MemoryManagerIsReady)
491 //        throw new InvalidOperationException("Менеджер памяти ещё не готов.");
492 //    if (LinkDoesNotExist(link))
493 //        throw new InvalidOperationException("Если связь не существует, её нельзя
494 //        ↪ заменить.");
495 //    if (LinkDoesNotExist(replacement))
496 //        throw new ArgumentException("Пустая или удалённая связь не может быть
497 //        ↪ замещаемым значением.", "replacement");
498 //    link = ReplaceLink(link, replacement);
499 //}
500
501 public static Link Search(Link source, Link linker, Link target)
502 {
503     if (_memoryManagerIsReady == default)
504     {
505         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
506     }
507     if (LinkDoesNotExist(source) || LinkDoesNotExist(linker) || LinkDoesNotExist(target))
508     {
509         throw new InvalidOperationException("Выполнить поиск связи можно только по
510         ↪ существующим связям.");
511     }
512     return SearchLink(source, linker, target);
513 }
514
515 public static bool Exists(Link source, Link linker, Link target) => SearchLink(source,
516     ↪ linker, target) != 0;
517
518 #endregion
519
520 #region Referers Walkers
521
522 public bool WalkThroughReferersAsSource(Func<Link, bool> walker)
523 {
524     if (LinkDoesNotExist(this))
525     {
526         throw new InvalidOperationException("С несуществующей связью нельзя
527         ↪ производить операции.");
528     }
529     var referers = ReferersBySourceCount;
530     if (referers == 1)
531     {
532         return walker(FirstRefererBySource);
533     }
534     else if (referers > 1)
535     {
536         return WalkThroughReferersBySource(this, x => walker(x) ? 1 : 0) != 0;
537     }
538 }

```

```

533     else
534     {
535         return true;
536     }
537 }
538
539 public void WalkThroughReferersAsSource(Action<Link> walker)
540 {
541     if (LinkDoesNotExist(this))
542     {
543         throw new InvalidOperationException("С несуществующей связью нельзя
544             ↪ производить операции.");
545     }
546     var referers = ReferersBySourceCount;
547     if (referers == 1)
548     {
549         walker(FirstRefererBySource);
550     }
551     else if (referers > 1)
552     {
553         WalkThroughAllReferersBySource(this, x => walker(x));
554     }
555 }
556
557 public bool WalkThroughReferersAsLinker(Func<Link, bool> walker)
558 {
559     if (LinkDoesNotExist(this))
560     {
561         throw new InvalidOperationException("С несуществующей связью нельзя
562             ↪ производить операции.");
563     }
564     var referers = ReferersByLinkerCount;
565     if (referers == 1)
566     {
567         return walker(FirstRefererByLinker);
568     }
569     else if (referers > 1)
570     {
571         return WalkThroughReferersByLinker(this, x => walker(x) ? 1 : 0) != 0;
572     }
573     else
574     {
575         return true;
576     }
577 }
578
579 public void WalkThroughReferersAsLinker(Action<Link> walker)
580 {
581     if (LinkDoesNotExist(this))
582     {
583         throw new InvalidOperationException("С несуществующей связью нельзя
584             ↪ производить операции.");
585     }
586     var referers = ReferersByLinkerCount;
587     if (referers == 1)
588     {
589         walker(FirstRefererByLinker);
590     }
591     else if (referers > 1)
592     {
593         WalkThroughAllReferersByLinker(this, x => walker(x));
594     }
595 }
596
597 public bool WalkThroughReferersAsTarget(Func<Link, bool> walker)
598 {
599     if (LinkDoesNotExist(this))
600     {
601         throw new InvalidOperationException("С несуществующей связью нельзя
602             ↪ производить операции.");
603     }
604     var referers = ReferersByTargetCount;
605     if (referers == 1)
606     {
607         return walker(FirstRefererByTarget);
608     }
609     else if (referers > 1)
610     {

```

```

607         return WalkThroughReferersByTarget(this, x => walker(x) ? 1 : 0) != 0;
608     }
609     else
610     {
611         return true;
612     }
613 }
614
615 public void WalkThroughReferersAsTarget(Action<Link> walker)
616 {
617     if (LinkDoesNotExist(this))
618     {
619         throw new InvalidOperationException("С несуществующей связью нельзя
        ↳ производить операции.");
620     }
621     var referers = ReferersByTargetCount;
622     if (referers == 1)
623     {
624         walker(FirstRefererByTarget);
625     }
626     else if (referers > 1)
627     {
628         WalkThroughAllReferersByTarget(this, x => walker(x));
629     }
630 }
631
632 public void WalkThroughReferers(Action<Link> walker)
633 {
634     if (LinkDoesNotExist(this))
635     {
636         throw new InvalidOperationException("С несуществующей связью нельзя
        ↳ производить операции.");
637     }
638     void wrapper(ulong x) => walker(x);
639     WalkThroughAllReferersBySource(this, wrapper);
640     WalkThroughAllReferersByLinker(this, wrapper);
641     WalkThroughAllReferersByTarget(this, wrapper);
642 }
643
644 public void WalkThroughReferers(Func<Link, bool> walker)
645 {
646     if (LinkDoesNotExist(this))
647     {
648         throw new InvalidOperationException("С несуществующей связью нельзя
        ↳ производить операции.");
649     }
650     long wrapper(ulong x) => walker(x) ? 1 : 0;
651     WalkThroughReferersBySource(this, wrapper);
652     WalkThroughReferersByLinker(this, wrapper);
653     WalkThroughReferersByTarget(this, wrapper);
654 }
655
656 public static bool WalkThroughAllLinks(Func<Link, bool> walker) => WalkThroughLinks(x =>
    ↳ walker(x) ? 1 : 0) != 0;
657
658 public static void WalkThroughAllLinks(Action<Link> walker) => WalkThroughAllLinks(new
    ↳ Visitor(x => walker(x)));
659
660 #endregion
661 }
662 }

```

1.6 ./csharp/Platform.Data.Triplets/LinkConverter.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.Sequences;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Triplets
8 {
9     public static class LinkConverter
10     {
11         public static Link FromList(List<Link> links)
12         {
13             var i = links.Count - 1;
14             var element = links[i];
15             while (--i >= 0)
16             {

```



```

17         element = links[i] & element;
18     }
19     return element;
20 }
21
22 public static Link FromList(Link[] links)
23 {
24     var i = links.Length - 1;
25     var element = links[i];
26     while (--i >= 0)
27     {
28         element = links[i] & element;
29     }
30     return element;
31 }
32
33 public static List<Link> ToList(Link link)
34 {
35     var list = new List<Link>();
36     SequenceWalker.WalkRight(link, x => x.Source, x => x.Target, x => x.Linker !=
        ↪ Net.And, list.Add);
37     return list;
38 }
39
40 public static Link FromNumber(long number) => NumberHelpers.FromNumber(number);
41
42 public static long ToNumber(Link number) => NumberHelpers.ToNumber(number);
43
44 public static Link FromChar(char c) => CharacterHelpers.FromChar(c);
45
46 public static char ToChar(Link charLink) => CharacterHelpers.ToChar(charLink);
47
48 public static Link FromChars(char[] chars) => FromObjectsToSequence(chars, FromChar);
49
50 public static Link FromChars(char[] chars, int takeFrom, int takeUntil) =>
    ↪ FromObjectsToSequence(chars, takeFrom, takeUntil, FromChar);
51
52 public static Link FromNumbers(long[] numbers) => FromObjectsToSequence(numbers,
    ↪ FromNumber);
53
54 public static Link FromNumbers(long[] numbers, int takeFrom, int takeUntil) =>
    ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, FromNumber);
55
56 public static Link FromNumbers(ushort[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪ FromNumber(x));
57
58 public static Link FromNumbers(ushort[] numbers, int takeFrom, int takeUntil) =>
    ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));
59
60 public static Link FromNumbers(uint[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪ FromNumber(x));
61
62 public static Link FromNumbers(uint[] numbers, int takeFrom, int takeUntil) =>
    ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));
63
64 public static Link FromNumbers(byte[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪ FromNumber(x));
65
66 public static Link FromNumbers(byte[] numbers, int takeFrom, int takeUntil) =>
    ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));
67
68 public static Link FromNumbers(bool[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪ FromNumber(x ? 1 : 0));
69
70 public static Link FromNumbers(bool[] numbers, int takeFrom, int takeUntil) =>
    ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x ? 1 : 0));
71
72 public static Link FromObjectsToSequence<T>(T[] objects, Func<T, Link> converter) =>
    ↪ FromObjectsToSequence(objects, 0, objects.Length, converter);
73
74 public static Link FromObjectsToSequence<T>(T[] objects, int takeFrom, int takeUntil,
    ↪ Func<T, Link> converter)
75 {
76     var length = takeUntil - takeFrom;
77     if (length <= 0)
78     {
79         throw new ArgumentOutOfRangeException(nameof(takeUntil), "Нельзя преобразовать
            ↪ пустой список к связям.");

```

```

80     }
81     var copy = new Link[length];
82     for (int i = takeFrom, j = 0; i < takeUntil; i++, j++)
83     {
84         copy[j] = converter(objects[i]);
85     }
86     return FromList(copy);
87 }
88
89 public static Link FromChars(string str)
90 {
91     var copy = new Link[str.Length];
92     for (var i = 0; i < copy.Length; i++)
93     {
94         copy[i] = FromChar(str[i]);
95     }
96     return FromList(copy);
97 }
98
99 public static Link FromString(string str)
100 {
101     var copy = new Link[str.Length];
102     for (var i = 0; i < copy.Length; i++)
103     {
104         copy[i] = FromChar(str[i]);
105     }
106     var strLink = Link.Create(Net.String, Net.ThatConsistsOf, FromList(copy));
107     return strLink;
108 }
109
110 public static string ToString(Link link)
111 {
112     if (link.IsString())
113     {
114         return ToString(ToList(link.Target));
115     }
116     throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
    ↪ string.");
117 }
118
119 public static string ToString(List<Link> charLinks)
120 {
121     var chars = new char[charLinks.Count];
122     for (var i = 0; i < charLinks.Count; i++)
123     {
124         chars[i] = ToChar(charLinks[i]);
125     }
126     return new string(chars);
127 }
128 }
129 }

```

1.7 ./csharp/Platform.Data.Triplets/LinkExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Data.Sequences;
5  using Platform.Data.Triplets.Sequences;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Triplets
10 {
11     public static class LinkExtensions
12     {
13         public static Link SetName(this Link link, string name)
14         {
15             Link.Create(link, Net.Has, Link.Create(Net.Name, Net.ThatIsRepresentedBy,
    ↪ LinkConverter.FromString(name)));
16             return link; // Chaining
17         }
18
19         private static readonly HashSet<Link> _linksWithNamesGatheringProcess = new
    ↪ HashSet<Link>();
20
21         public static bool TryGetName(this Link link, out string str)
22         {
23             // Защита от заикливания
24             if (!_linksWithNamesGatheringProcess.Add(link))

```

```

25 {
26     str = "...";
27     return true;
28 }
29 try
30 {
31     if (link != null)
32     {
33         if (link.Linker == Net.And)
34         {
35             str = SequenceHelpers.FormatSequence(link);
36             return true;
37         }
38         else if (link.IsGroup())
39         {
40             str = LinkConverter.ToString(LinkConverter.ToList(link.Target));
41             return true;
42         }
43         else if (link.IsChar())
44         {
45             str = LinkConverter.ToChar(link).ToString();
46             return true;
47         }
48         else if (link.TryGetSpecificName(out str))
49         {
50             return true;
51         }
52
53         if (link.Source == link || link.Linker == link || link.Target == link)
54         {
55             return false;
56         }
57
58         if (link.Source.TryGetName(out string sourceName) &&
59             ↪ link.Linker.TryGetName(out string linkerName) &&
60             ↪ link.Target.TryGetName(out string targetName))
61         {
62             var sb = new StringBuilder();
63             sb.Append(sourceName).Append(' ').Append(linkerName).Append(' '
64             ↪ ').Append(targetName);
65             str = sb.ToString();
66             return true;
67         }
68     }
69     str = null;
70     return false;
71 }
72 finally
73 {
74     _linksWithNamesGatheringProcess.Remove(link);
75 }
76
77 public static bool TryGetSpecificName(this Link link, out string name)
78 {
79     string nameLocal = null;
80     if (Net.Name.ReferersBySourceCount < link.ReferersBySourceCount)
81     {
82         Net.Name.WalkThroughReferersAsSource(referer =>
83         {
84             if (referer.Linker == Net.ThatIsRepresentedBy)
85             {
86                 if (Link.Exists(link, Net.Has, referer))
87                 {
88                     nameLocal = LinkConverter.ToString(referer.Target);
89                     return false; // Останавливаем проход
90                 }
91             }
92             return true;
93         });
94     }
95     else
96     {
97         link.WalkThroughReferersAsSource(referer =>
98         {
99             if (referer.Linker == Net.Has)
100             {
101                 var nameLink = referer.Target;

```

```

100         if (nameLink.Source == Net.Name && nameLink.Linker ==
101             ↪ Net.ThatIsRepresentedBy)
102         {
103             nameLocal = LinkConverter.ToString(nameLink.Target);
104             return false; // Останавливаем проход
105         }
106         return true;
107     });
108 }
109
110 name = nameLocal;
111 return nameLocal != null;
112 }
113
114 // Проверка на принадлежность классу
115 public static bool Is(this Link link, Link @class)
116 {
117     if (link.Linker == Net.IsA)
118     {
119         if (link.Target == @class)
120         {
121             return true;
122         }
123         else
124         {
125             return link.Target.Is(@class);
126         }
127     }
128     return false;
129 }
130
131 // Несколько не правильное определение, так выйдет, что любая сумма входящая в диапазон
132 ↪ значений char будет символом.
133 // Нужно изменить определение чара, идеально: char consists of sum of [8, 64].
134 public static bool IsChar(this Link link) => CharacterHelpers.IsChar(link);
135
136 public static bool IsGroup(this Link link) => link != null && link.Source == Net.Group
137 ↪ && link.Linker == Net.ThatConsistsOf;
138
139 public static bool IsSum(this Link link) => link != null && link.Source == Net.Sum &&
140 ↪ link.Linker == Net.Of;
141
142 public static bool IsString(this Link link) => link != null && link.Source == Net.String
143 ↪ && link.Linker == Net.ThatConsistsOf;
144
145 public static bool IsName(this Link link) => link != null && link.Source == Net.Name &&
146 ↪ link.Linker == Net.Of;
147
148 public static Link[] GetArrayOfRerersBySource(this Link link)
149 {
150     if (link == null)
151     {
152         return new Link[0];
153     }
154     else
155     {
156         var array = new Link[link.ReferersBySourceCount];
157         var index = 0;
158         link.WalkThroughReferersAsSource(referer => array[index++] = referer);
159         return array;
160     }
161 }
162
163 public static Link[] GetArrayOfRerersByLinker(this Link link)
164 {
165     if (link == null)
166     {
167         return new Link[0];
168     }
169     else
170     {
171         var array = new Link[link.ReferersByLinkerCount];
172         var index = 0;
173         link.WalkThroughReferersAsLinker(referer => array[index++] = referer);
174         return array;
175     }
176 }

```

```

173 public static Link[] GetArrayOfRererersByTarget(this Link link)
174 {
175     if (link == null)
176     {
177         return new Link[0];
178     }
179     else
180     {
181         var array = new Link[link.ReferersByTargetCount];
182         var index = 0;
183         link.WalkThroughReferersAsTarget(referer => array[index++] = referer);
184         return array;
185     }
186 }
187
188 public static void WalkThroughSequence(this Link link, Action<Link> action) =>
    ↳ SequenceWalker.WalkRight(link, x => x.Source, x => x.Target, x => x.Linker !=
    ↳ Net.And, action);
189 }
190 }

```

1.8 ./csharp/Platform.Data.Triplets/Net.cs

```

1 using Platform.Threading;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Triplets
6 {
7     public enum NetMapping : long
8     {
9         Link,
10        Thing,
11        IsA,
12        IsNotA,
13
14        Of,
15        And,
16        ThatConsistsOf,
17        Has,
18        Contains,
19        ContainedBy,
20
21        One,
22        Zero,
23
24        Sum,
25        Character,
26        String,
27        Name,
28
29        Set,
30        Group,
31
32        ParsedFrom,
33        ThatIs,
34        ThatIsBefore,
35        ThatIsBetween,
36        ThatIsAfter,
37        ThatIsRepresentedBy,
38        ThatHas,
39
40        Text,
41        Path,
42        Content,
43        EmptyContent,
44        Empty,
45        Alphabet,
46        Letter,
47        Case,
48        Upper,
49        UpperCase,
50        Lower,
51        LowerCase,
52        Code
53    }
54
55    public static class Net
56    {
57        public static Link Link { get; private set; }
58        public static Link Thing { get; private set; }
59        public static Link IsA { get; private set; }
60        public static Link IsNotA { get; private set; }

```

```

61
62 public static Link Of { get; private set; }
63 public static Link And { get; private set; }
64 public static Link ThatConsistsOf { get; private set; }
65 public static Link Has { get; private set; }
66 public static Link Contains { get; private set; }
67 public static Link ContainedBy { get; private set; }
68
69 public static Link One { get; private set; }
70 public static Link Zero { get; private set; }
71
72 public static Link Sum { get; private set; }
73 public static Link Character { get; private set; }
74 public static Link String { get; private set; }
75 public static Link Name { get; private set; }
76
77 public static Link Set { get; private set; }
78 public static Link Group { get; private set; }
79
80 public static Link ParsedFrom { get; private set; }
81 public static Link ThatIs { get; private set; }
82 public static Link ThatIsBefore { get; private set; }
83 public static Link ThatIsBetween { get; private set; }
84 public static Link ThatIsAfter { get; private set; }
85 public static Link ThatIsRepresentedBy { get; private set; }
86 public static Link ThatHas { get; private set; }
87
88 public static Link Text { get; private set; }
89 public static Link Path { get; private set; }
90 public static Link Content { get; private set; }
91 public static Link EmptyContent { get; private set; }
92 public static Link Empty { get; private set; }
93 public static Link Alphabet { get; private set; }
94 public static Link Letter { get; private set; }
95 public static Link Case { get; private set; }
96 public static Link Upper { get; private set; }
97 public static Link UpperCase { get; private set; }
98 public static Link Lower { get; private set; }
99 public static Link LowerCase { get; private set; }
100 public static Link Code { get; private set; }
101
102 static Net() => Create();
103
104 public static Link CreateThing() => Link.Create(Link.Itself, IsA, Thing);
105
106 public static Link CreateMappedThing(object mapping) => Link.CreateMapped(Link.Itself,
    ↪ IsA, Thing, mapping);
107
108 public static Link CreateLink() => Link.Create(Link.Itself, IsA, Link);
109
110 public static Link CreateMappedLink(object mapping) => Link.CreateMapped(Link.Itself,
    ↪ IsA, Link, mapping);
111
112 public static Link CreateSet() => Link.Create(Link.Itself, IsA, Set);
113
114 private static void Create()
115 {
116     #region Core
117
118     IsA = Link.GetMappedOrDefault(NetMapping.IsA);
119     IsNotA = Link.GetMappedOrDefault(NetMapping.IsNotA);
120     Link = Link.GetMappedOrDefault(NetMapping.Link);
121     Thing = Link.GetMappedOrDefault(NetMapping.Thing);
122
123     if (IsA == null || IsNotA == null || Link == null || Thing == null)
124     {
125         // Наивная инициализация (Не является корректным объяснением).
126         IsA = Link.CreateMapped(Link.Itself, Link.Itself, Link.Itself, NetMapping.IsA);
127         ↪ // Строит переделывать в "[x] is a member|instance|element of the class [y]"
128         IsNotA = Link.CreateMapped(Link.Itself, Link.Itself, IsA, NetMapping.IsNotA);
129         Link = Link.CreateMapped(Link.Itself, IsA, Link.Itself, NetMapping.Link);
130         Thing = Link.CreateMapped(Link.Itself, IsNotA, Link, NetMapping.Thing);
131
132         IsA = Link.Update(IsA, IsA, IsA, Link); // Исключение, позволяющие завершить
133         ↪ систему
134     }
135     #endregion

```

```

136 Of = CreateMappedLink(NetMapping.Of);
137 And = CreateMappedLink(NetMapping.And);
138 ThatConsistsOf = CreateMappedLink(NetMapping.ThatConsistsOf);
139 Has = CreateMappedLink(NetMapping.Has);
140 Contains = CreateMappedLink(NetMapping.Contains);
141 ContainedBy = CreateMappedLink(NetMapping.ContainedBy);
142
143 One = CreateMappedThing(NetMapping.One);
144 Zero = CreateMappedThing(NetMapping.Zero);
145
146 Sum = CreateMappedThing(NetMapping.Sum);
147 Character = CreateMappedThing(NetMapping.Character);
148 String = CreateMappedThing(NetMapping.String);
149 Name = Link.CreateMapped(Link.Itself, IsA, String, NetMapping.Name);
150
151 Set = CreateMappedThing(NetMapping.Set);
152 Group = CreateMappedThing(NetMapping.Group);
153
154 ParsedFrom = CreateMappedLink(NetMapping.ParsedFrom);
155 ThatIs = CreateMappedLink(NetMapping.ThatIs);
156 ThatIsBefore = CreateMappedLink(NetMapping.ThatIsBefore);
157 ThatIsAfter = CreateMappedLink(NetMapping.ThatIsAfter);
158 ThatIsBetween = CreateMappedLink(NetMapping.ThatIsBetween);
159 ThatIsRepresentedBy = CreateMappedLink(NetMapping.ThatIsRepresentedBy);
160 ThatHas = CreateMappedLink(NetMapping.ThatHas);
161
162 Text = CreateMappedThing(NetMapping.Text);
163 Path = CreateMappedThing(NetMapping.Path);
164 Content = CreateMappedThing(NetMapping.Content);
165 Empty = CreateMappedThing(NetMapping.Empty);
166 EmptyContent = Link.CreateMapped(Content, ThatIs, Empty, NetMapping.EmptyContent);
167 Alphabet = CreateMappedThing(NetMapping.Alphabet);
168 Letter = Link.CreateMapped(Link.Itself, IsA, Character, NetMapping.Letter);
169 Case = CreateMappedThing(NetMapping.Case);
170 Upper = CreateMappedThing(NetMapping.Upper);
171 UpperCase = Link.CreateMapped(Case, ThatIs, Upper, NetMapping.UpperCase);
172 Lower = CreateMappedThing(NetMapping.Lower);
173 LowerCase = Link.CreateMapped(Case, ThatIs, Lower, NetMapping.LowerCase);
174 Code = CreateMappedThing(NetMapping.Code);
175
176 SetNames();
177 }
178
179 public static void Recreate()
180 {
181     ThreadHelpers.InvokeWithExtendedMaxStackSize(() => Link.Delete(IsA));
182     CharacterHelpers.Recreate();
183     Create();
184 }
185
186 private static void SetNames()
187 {
188     Thing.SetName("thing");
189     Link.SetName("link");
190     IsA.SetName("is a");
191     IsNotA.SetName("is not a");
192
193     Of.SetName("of");
194     And.SetName("and");
195     ThatConsistsOf.SetName("that consists of");
196     Has.SetName("has");
197     Contains.SetName("contains");
198     ContainedBy.SetName("contained by");
199
200     One.SetName("one");
201     Zero.SetName("zero");
202
203     Character.SetName("character");
204     Sum.SetName("sum");
205     String.SetName("string");
206     Name.SetName("name");
207
208     Set.SetName("set");
209     Group.SetName("group");
210
211     ParsedFrom.SetName("parsed from");
212     ThatIs.SetName("that is");
213     ThatIsBefore.SetName("that is before");
214     ThatIsAfter.SetName("that is after");

```

```

215     ThatIsBetween.SetName("that is between");
216     ThatIsRepresentedBy.SetName("that is represented by");
217     ThatHas.SetName("that has");
218
219     Text.SetName("text");
220     Path.SetName("path");
221     Content.SetName("content");
222     Empty.SetName("empty");
223     EmptyContent.SetName("empty content");
224     Alphabet.SetName("alphabet");
225     Letter.SetName("letter");
226     Case.SetName("case");
227     Upper.SetName("upper");
228     Lower.SetName("lower");
229     Code.SetName("code");
230 }
231 }
232 }

```

1.9 ./csharp/Platform.Data.Triplets/NumberHelpers.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Triplets
9  {
10     public static class NumberHelpers
11     {
12         public static Link[] NumbersToLinks { get; private set; }
13         public static Dictionary<Link, long> LinksToNumbers { get; private set; }
14
15         static NumberHelpers() => Create();
16
17         private static void Create()
18         {
19             NumbersToLinks = new Link[64];
20             LinksToNumbers = new Dictionary<Link, long>();
21             NumbersToLinks[0] = Net.One;
22             LinksToNumbers[Net.One] = 1;
23         }
24
25         public static void Recreate() => Create();
26
27         private static Link FromPowerOf2(long powerOf2)
28         {
29             var result = NumbersToLinks[powerOf2];
30             if (result == null)
31             {
32                 var previousPowerOf2Link = NumbersToLinks[powerOf2 - 1];
33                 if (previousPowerOf2Link == null)
34                 {
35                     previousPowerOf2Link = NumbersToLinks[0];
36                     for (var i = 1; i < powerOf2; i++)
37                     {
38                         if (NumbersToLinks[i] == null)
39                         {
40                             var numberLink = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
41                                 ↪ previousPowerOf2Link);
42                             var num = (long)System.Math.Pow(2, i);
43                             NumbersToLinks[i] = numberLink;
44                             LinksToNumbers[numberLink] = num;
45                             numberLink.SetName(num.ToString(CultureInfo.InvariantCulture));
46                         }
47                         previousPowerOf2Link = NumbersToLinks[i];
48                     }
49                     result = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
50                         ↪ previousPowerOf2Link);
51                     var number = (long)System.Math.Pow(2, powerOf2);
52                     NumbersToLinks[powerOf2] = result;
53                     LinksToNumbers[result] = number;
54                     result.SetName(number.ToString(CultureInfo.InvariantCulture));
55                 }
56                 return result;
57             }
58         }
59     }
60 }

```



```

58 public static Link FromNumber(long number)
59 {
60     if (number == 0)
61     {
62         return Net.Zero;
63     }
64     if (number == 1)
65     {
66         return Net.One;
67     }
68     var links = new Link[Bit.Count(number)];
69     if (number >= 0)
70     {
71         for (long key = 1, powerOf2 = 0, i = 0; key <= number; key *= 2, powerOf2++)
72         {
73             if ((number & key) == key)
74             {
75                 links[i] = FromPowerOf2(powerOf2);
76                 i++;
77             }
78         }
79     }
80     else
81     {
82         throw new NotSupportedException("Negative numbers are not supported yet.");
83     }
84     var sum = Link.Create(Net.Sum, Net.Of, LinkConverter.FromList(links));
85     return sum;
86 }
87
88 public static long ToNumber(Link link)
89 {
90     if (link == Net.Zero)
91     {
92         return 0;
93     }
94     if (link == Net.One)
95     {
96         return 1;
97     }
98     if (link.IsSum())
99     {
100         var numberParts = LinkConverter.ToList(link.Target);
101         long number = 0;
102         for (var i = 0; i < numberParts.Count; i++)
103         {
104             GoDownAndTakeIt(numberParts[i], out long numberPart);
105             number += numberPart;
106         }
107         return number;
108     }
109     throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
    ↪ number.");
110 }
111
112 private static void GoDownAndTakeIt(Link link, out long number)
113 {
114     if (!LinksToNumbers.TryGetValue(link, out number))
115     {
116         var previousNumberLink = link.Target.Source;
117         GoDownAndTakeIt(previousNumberLink, out number);
118         var previousNumberIndex = (int)System.Math.Log(number, 2);
119         var newNumberIndex = previousNumberIndex + 1;
120         var newNumberLink = Link.Create(Net.Sum, Net.Of, previousNumberLink &
    ↪ previousNumberLink);
121         number += number;
122         NumbersToLinks[newNumberIndex] = newNumberLink;
123         LinksToNumbers[newNumberLink] = number;
124     }
125 }
126 }
127 }

```

1.10 ./csharp/Platform.Data.Triplets/Sequences/CompressionExperiments.cs

```

1 using System;
2 using System.Collections.Generic;
3
4 namespace Platform.Data.Triplets.Sequences
5 {

```

```

6 internal static class CompressionExperiments
7 {
8     public static void RightJoin(ref Link subject, Link @object)
9     {
10         if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
11             ↪ subject.ReferersByTargetCount == 0)
12         {
13             var subJoint = Link.Search(subject.Target, Net.And, @object);
14             if (subJoint != null && subJoint != subject)
15             {
16                 Link.Update(ref subject, subject.Source, Net.And, subJoint);
17                 return;
18             }
19             subject = Link.Create(subject, Net.And, @object);
20         }
21
22         //public static Link RightJoinUnsafe(Link subject, Link @object)
23         //{
24         //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
25         //        ↪ subject.ReferersByTargetCount == 0)
26         //    {
27         //        Link subJoint = Link.Search(subject.Target, Net.And, @object);
28         //        if (subJoint != null && subJoint != subject)
29         //        {
30         //            Link.Update(ref subject, subject.Source, Net.And, subJoint);
31         //            return subject;
32         //        }
33         //    }
34         //    return Link.Create(subject, Net.And, @object);
35         //}
36
37         ///public static void LeftJoin(ref Link subject, Link @object)
38         ///{
39         ///    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
40         ///        ↪ subject.ReferersByTargetCount == 0)
41         ///    {
42         ///        Link subJoint = Link.Search(@object, Net.And, subject.Source);
43         ///        if (subJoint != null && subJoint != subject)
44         ///        {
45         ///            Link.Update(ref subject, subJoint, Net.And, subject.Target);
46         ///            return;
47         ///        }
48         ///    }
49         ///    subject = Link.Create(@object, Net.And, subject);
50         ///}
51
52     public static void LeftJoin(ref Link subject, Link @object)
53     {
54         if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
55             ↪ subject.ReferersByTargetCount == 0)
56         {
57             var subJoint = Link.Search(@object, Net.And, subject.Source);
58             if (subJoint != null && subJoint != subject)
59             {
60                 Link.Update(ref subject, subJoint, Net.And, subject.Target);
61                 //var prev = Link.Search(@object, Net.And, subject);
62                 //if (prev != null)
63                 //{
64                     Link.Update(ref prev, subJoint, Net.And, subject.Target);
65                 //}
66                 return;
67             }
68         }
69         subject = Link.Create(@object, Net.And, subject);
70     }
71
72     // Сначала сжатие налево, а затем направо (так эффективнее)
73     // Не приятный момент, что обе связи, и первая и вторая могут быть изменены в результате
74     // ↪ алгоритма.
75     //public static Link CombinedJoin(ref Link first, ref Link second)
76     //{
77     //    Link atomicConnection = Link.Search(first, Net.And, second);
78     //    if (atomicConnection != null)
79     //    {
80         return atomicConnection;
81     //    }
82     //    else

```

```

79 // {
80 //     if (second.Linker == Net.And)
81 //     {
82 //         Link subJoint = Link.Search(first, Net.And, second.Source);
83 //         if (subJoint != null && subJoint != second)// && subJoint.TotalReferers >
↪ second.TotalReferers)
84 //         {
85 //             //if (first.Linker == Net.And)
86 //             //{
87 //                 // TODO: ...
88 //             //}
89 //             if (second.TotalReferers > 0)
90 //             {
91 //                 // В данный момент это никак не влияет, из-за того что добавлено
↪ условие по требованию
92 //                 // использования атомарного соедининения если оно есть
93
94 //                 // В целом же приоритет между обходным соединением и атомарным
↪ нужно определять по весу.
95 //                 // И если в сети обнаружено сразу два варианта прохода - простой и
↪ обходной - нужно перебрасывать
96 //                 // пути с меньшим весом на использование путей с большим весом.
↪ (Это и технически эффективнее и более оправдано
97 //                 // с точки зрения смысла).
98
99 //                 // Положительный эффект текущей реализации, что она быстро
↪ "успокаивается" набирает критическую массу
100 //                 // и перестаёт вести себя не предсказуемо
101
102 //                 // Неприятность учёта веса в том, что нужно обрабатывать большое
↪ количество комбинаций.
103 //                 // Но вероятно это оправдано.
104
105 //                 //var prev = Link.Search(first, Net.And, second);
106 //                 //if (prev != null && subJoint != prev) // && prev.TotalReferers <
↪ subJoint.TotalReferers)
107 //                 //{
108 //                     // Link.Update(ref prev, subJoint, Net.And, second.Target);
109 //                     // if (second.TotalReferers == 0)
110 //                     // {
111 //                         // Link.Delete(ref second);
112 //                     // }
113 //                     // return prev;
114 //                 //}
115 //                 //return Link.Create(subJoint, Net.And, second.Target);
116 //             }
117 //         else
118 //         {
119 //             Link.Update(ref second, subJoint, Net.And, second.Target);
120 //             return second;
121 //         }
122 //     }
123 // }
124 // if (first.Linker == Net.And)
125 // {
126 //     Link subJoint = Link.Search(first.Target, Net.And, second);
127 //     if (subJoint != null && subJoint != first)// && subJoint.TotalReferers >
↪ first.TotalReferers)
128 //     {
129 //         if (first.TotalReferers > 0)
130 //         {
131 //             //var prev = Link.Search(first, Net.And, second);
132 //             //if (prev != null && subJoint != prev) // && prev.TotalReferers <
↪ subJoint.TotalReferers)
133 //             //{
134 //                 // Link.Update(ref prev, first.Source, Net.And, subJoint);
135 //                 // if (first.TotalReferers == 0)
136 //                 // {
137 //                     // Link.Delete(ref first);
138 //                 // }
139 //                 // return prev;
140 //             //}
141 //             //return Link.Create(first.Source, Net.And, subJoint);
142 //         }
143 //     else
144 //     {
145 //         Link.Update(ref first, first.Source, Net.And, subJoint);

```

```

146 //          return first;
147 //      }
148 //  }
149 //      }
150 //      return Link.Create(first, Net.And, second);
151 //  }
152 //}
153
154 public static int CompressionsCount;
155
156 public static Link CombinedJoin(ref Link first, ref Link second)
157 {
158     // Перестроение работает хорошо только когда одна из связей является парой и
159     // ↳ аккумулятором одновременно
160     // Когда обе связи - пары - нужно использовать другой алгоритм, иначе сжатие будет
161     // ↳ отсутствовать.
162     //if ((first.Linker == Net.And && second.Linker != Net.And)
163     // || (second.Linker == Net.And && first.Linker != Net.And))
164     //{
165     //    Link connection = TryReconstructConnection(first, second);
166     //    if (connection != null)
167     //    {
168     //        CompressionsCount++;
169     //        return connection;
170     //    }
171     //}
172     //return first & second;
173     //long totalDoublets = Net.And.ReferersByLinkerCount;
174     if (first == null || second == null)
175     {
176     }
177     var directConnection = Link.Search(first, Net.And, second);
178     if (directConnection == null)
179     {
180         directConnection = TryReconstructConnection(first, second);
181     }
182     Link rightCrossConnection = null;
183     if (second.Linker == Net.And)
184     {
185         var assumedRightCrossConnection = Link.Search(first, Net.And, second.Source);
186         if (assumedRightCrossConnection != null && second != assumedRightCrossConnection)
187         {
188             rightCrossConnection = assumedRightCrossConnection;
189         }
190         else
191         {
192             rightCrossConnection = TryReconstructConnection(first, second.Source);
193         }
194     }
195     Link leftCrossConnection = null;
196     if (first.Linker == Net.And)
197     {
198         var assumedLeftCrossConnection = Link.Search(first.Target, Net.And, second);
199         if (assumedLeftCrossConnection != null && first != assumedLeftCrossConnection)
200         {
201             leftCrossConnection = assumedLeftCrossConnection;
202         }
203         else
204         {
205             leftCrossConnection = TryReconstructConnection(first.Target, second);
206         }
207     }
208     // Наверное имеет смысл только в "безвыходной" ситуации
209     //if (directConnection == null && rightCrossConnection == null &&
210     // ↳ leftCrossConnection == null)
211     //{
212     //    directConnection = TryReconstructConnection(first, second);
213     //    // Может давать более агрессивное сжатие, но теряется стабильность
214     //    //if (directConnection == null)
215     //    //{
216     //        //if (second.Linker == Net.And)
217     //        //{
218     //            Link assumedRightCrossConnection = TryReconstructConnection(first,
219     // ↳ second.Source);
220     //            //if (assumedRightCrossConnection != null && second !=
221     // ↳ assumedRightCrossConnection)
222     //            {
223     //                rightCrossConnection = assumedRightCrossConnection;
224     //            }
225     //        }
226     //    }
227     //}

```

```

219 // // // }
220 // // //}
221 // // //if (rightCrossConnection == null)
222 // // //{
223 // // //if (first.Linker == Net.And)
224 // // //{
225 // // //    Link assumedLeftCrossConnection =
226 ↪ TryReconstructConnection(first.Target, second);
227 // // //    if (assumedLeftCrossConnection != null && first !=
228 ↪ assumedLeftCrossConnection)
229 // // //    {
230 // // //        leftCrossConnection = assumedLeftCrossConnection;
231 // // //    }
232 // // //}
233 // // //}
234 //Link middleCrossConnection = null;
235 //if (second.Linker == Net.And && first.Linker == Net.And)
236 //{
237 //    Link assumedMiddleCrossConnection = Link.Search(first.Target, Net.And,
238 ↪ second.Source);
239 //    if (assumedMiddleCrossConnection != null && first !=
240 ↪ assumedMiddleCrossConnection && second != assumedMiddleCrossConnection)
241 //    {
242 //        middleCrossConnection = assumedMiddleCrossConnection;
243 //    }
244 //}
245 //Link rightMiddleCrossConnection = null;
246 //if (middleCrossConnection != null)
247 //{
248 //}
249 if (directConnection != null
250 && (rightCrossConnection == null || directConnection.TotalReferers >=
251 ↪ rightCrossConnection.TotalReferers)
252 && (leftCrossConnection == null || directConnection.TotalReferers >=
253 ↪ leftCrossConnection.TotalReferers))
254 {
255     if (rightCrossConnection != null)
256     {
257         var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
258         if (prev != null && directConnection != prev)
259         {
260             Link.Update(ref prev, first, Net.And, second);
261         }
262         if (rightCrossConnection.TotalReferers == 0)
263         {
264             Link.Delete(ref rightCrossConnection);
265         }
266     }
267     if (leftCrossConnection != null)
268     {
269         var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
270         if (prev != null && directConnection != prev)
271         {
272             Link.Update(ref prev, first, Net.And, second);
273         }
274         if (leftCrossConnection.TotalReferers == 0)
275         {
276             Link.Delete(ref leftCrossConnection);
277         }
278     }
279     TryReconstructConnection(first, second);
280     return directConnection;
281 }
282 else if (rightCrossConnection != null
283 && (directConnection == null || rightCrossConnection.TotalReferers >=
284 ↪ directConnection.TotalReferers)
285 && (leftCrossConnection == null || rightCrossConnection.TotalReferers >=
286 ↪ leftCrossConnection.TotalReferers))
287 {
288     if (directConnection != null)
289     {
290         var prev = Link.Search(first, Net.And, second);
291         if (prev != null && rightCrossConnection != prev)
292         {
293             Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);

```

```

288     }
289 }
290 if (leftCrossConnection != null)
291 {
292     var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
293     if (prev != null && rightCrossConnection != prev)
294     {
295         Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
296     }
297 }
298 //TryReconstructConnection(first, second.Source);
299 //TryReconstructConnection(rightCrossConnection, second.Target); // ухудшает
    ↳ стабильность
300 var resultConnection = rightCrossConnection & second.Target;
301 //if (second.TotalReferers == 0)
302 //    Link.Delete(ref second);
303 return resultConnection;
304 }
305 else if (leftCrossConnection != null
306     && (directConnection == null || leftCrossConnection.TotalReferers >=
    ↳ directConnection.TotalReferers)
307     && (rightCrossConnection == null || leftCrossConnection.TotalReferers >=
    ↳ rightCrossConnection.TotalReferers))
308 {
309     if (directConnection != null)
310     {
311         var prev = Link.Search(first, Net.And, second);
312         if (prev != null && leftCrossConnection != prev)
313         {
314             Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
315         }
316     }
317     if (rightCrossConnection != null)
318     {
319         var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
320         if (prev != null && rightCrossConnection != prev)
321         {
322             Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
323         }
324     }
325     //TryReconstructConnection(first.Target, second);
326     //TryReconstructConnection(first.Source, leftCrossConnection); // ухудшает
    ↳ стабильность
327 var resultConnection = first.Source & leftCrossConnection;
328 //if (first.TotalReferers == 0)
329 //    Link.Delete(ref first);
330 return resultConnection;
331 }
332 else
333 {
334     if (directConnection != null)
335     {
336         return directConnection;
337     }
338     if (rightCrossConnection != null)
339     {
340         return rightCrossConnection & second.Target;
341     }
342     if (leftCrossConnection != null)
343     {
344         return first.Source & leftCrossConnection;
345     }
346 }
347 // Можно фиксировать по окончании каждой из веток, какой эффект от неё происходит
    ↳ (на сколько уменьшается/увеличивается количество связей)
348 directConnection = first & second;
349 //long difference = Net.And.ReferersByLinkerCount - totalDoublets;
350 //if (difference != 1)
351 //{
352 //    Console.WriteLine(Net.And.ReferersByLinkerCount - totalDoublets);
353 //}
354 return directConnection;
355 }
356
357 private static Link TryReconstructConnection(Link first, Link second)
358 {
359     Link directConnection = null;
360     if (second.ReferersBySourceCount < first.ReferersBySourceCount)

```

```

361 {
362     // o_|      x_o ...
363     // x_|      |___|
364     //
365     // <-
366     second.WalkThroughReferersAsSource(couple =>
367     {
368         if (couple.Linker == Net.And && couple.ReferersByTargetCount == 1 &&
369             ↪ couple.ReferersBySourceCount == 0)
370         {
371             var neighbour = couple.FirstRefererByTarget;
372             if (neighbour.Linker == Net.And && neighbour.Source == first)
373             {
374                 if (directConnection == null)
375                 {
376                     directConnection = first & second;
377                 }
378                 Link.Update(ref neighbour, directConnection, Net.And, couple.Target);
379                 //Link.Delete(ref couple); // Можно заменить удалением couple
380             }
381         }
382         if (couple.Linker == Net.And)
383         {
384             var neighbour = couple.FirstRefererByTarget;
385             if (neighbour.Linker == Net.And && neighbour.Source == first)
386             {
387                 throw new NotImplementedException();
388             }
389         }
390     });
391 else
392 {
393     // o_|      x_o ...
394     // x_|      |___|
395     //
396     // ->
397     first.WalkThroughReferersAsSource(couple =>
398     {
399         if (couple.Linker == Net.And)
400         {
401             var neighbour = couple.Target;
402             if (neighbour.Linker == Net.And && neighbour.Source == second)
403             {
404                 if (neighbour.ReferersByTargetCount == 1 &&
405                     ↪ neighbour.ReferersBySourceCount == 0)
406                 {
407                     if (directConnection == null)
408                     {
409                         directConnection = first & second;
410                     }
411                     Link.Update(ref couple, directConnection, Net.And,
412                         ↪ neighbour.Target);
413                     //Link.Delete(ref neighbour);
414                 }
415             }
416         }
417     });
418 }
419 if (second.ReferersByTargetCount < first.ReferersByTargetCount)
420 {
421     // |_x      ... x_o
422     // |_o      |___|
423     //
424     // <-
425     second.WalkThroughReferersAsTarget(couple =>
426     {
427         if (couple.Linker == Net.And)
428         {
429             var neighbour = couple.Source;
430             if (neighbour.Linker == Net.And && neighbour.Target == first)
431             {
432                 if (neighbour.ReferersByTargetCount == 0 &&
433                     ↪ neighbour.ReferersBySourceCount == 1)
434                 {
435                     if (directConnection == null)
436                     {

```

```

435         directConnection = first & second;
436     }
437     Link.Update(ref couple, neighbour.Source, Net.And,
438         ↪ directConnection);
439     //Link.Delete(ref neighbour);
440 }
441 }
442 });
443 }
444 else
445 {
446     // |_x      ... x_o
447     // |_o      |__|
448     //
449     // ->
450     first.WalkThroughReferersAsTarget((couple) =>
451     {
452         if (couple.Linker == Net.And && couple.ReferersByTargetCount == 0 &&
453             ↪ couple.ReferersBySourceCount == 1)
454         {
455             var neighbour = couple.FirstRefererBySource;
456             if (neighbour.Linker == Net.And && neighbour.Target == second)
457             {
458                 if (directConnection == null)
459                 {
460                     directConnection = first & second;
461                 }
462                 Link.Update(ref neighbour, couple.Source, Net.And, directConnection);
463                 Link.Delete(ref couple);
464             }
465         }
466     });
467     if (directConnection != null)
468     {
469         CompressionsCount++;
470     }
471     return directConnection;
472 }
473
474 ///public static Link CombinedJoin(Link left, Link right)
475 ///{
476 ///    Link rightSubJoint = Link.Search(left, Net.And, right.Source);
477 ///    if (rightSubJoint != null && rightSubJoint != right)
478 ///    {
479 ///        long rightSubJointReferers = rightSubJoint.TotalReferers;
480 ///        Link leftSubJoint = Link.Search(left.Target, Net.And, right);
481 ///        if (leftSubJoint != null && leftSubJoint != left)
482 ///        {
483 ///            long leftSubJointReferers = leftSubJoint.TotalReferers;
484 ///            if (leftSubJointReferers > rightSubJointReferers)
485 ///            {
486 ///                long leftReferers = left.TotalReferers;
487 ///                if (leftReferers > 0)
488 ///                {
489 ///                    return Link.Create(left.Source, Net.And, leftSubJoint);
490 ///                }
491 ///                else
492 ///                {
493 ///                    Link.Update(ref left, left.Source, Net.And, leftSubJoint);
494 ///                    return left;
495 ///                }
496 ///            }
497 ///        }
498 ///        long rightReferers = right.TotalReferers;
499 ///        if (rightReferers > 0)
500 ///        {
501 ///            return Link.Create(rightSubJoint, Net.And, right.Target);
502 ///        }
503 ///        else
504 ///        {
505 ///            Link.Update(ref right, rightSubJoint, Net.And, right.Target);
506 ///            return right;
507 ///        }
508 ///    }
509 ///    return Link.Create(left, Net.And, right);
510 ///}

```



```

511 //public static Link CombinedJoin(Link left, Link right)
512 //{
513 //    long leftReferers = left.TotalReferers;
514 //    Link leftSubJoint = Link.Search(left.Target, Net.And, right);
515 //    if (leftSubJoint != null && leftSubJoint != left)
516 //    {
517 //        long leftSubJointReferers = leftSubJoint.TotalReferers;
518 //    }
519 //    long rightReferers = left.TotalReferers;
520 //    Link rightSubJoint = Link.Search(left, Net.And, right.Source);
521 //    long rightSubJointReferers = rightSubJoint != null ? rightSubJoint.TotalReferers :
    ↳ long.MinValue;
522 //}
523 //public static Link LeftJoinUnsafe(Link subject, Link @object)
524 //{
525 //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
    ↳ subject.ReferersByTargetCount == 0)
526 //    {
527 //        Link subJoint = Link.Search(@object, Net.And, subject.Source);
528 //        if (subJoint != null && subJoint != subject)
529 //        {
530 //            Link.Update(ref subject, subJoint, Net.And, subject.Target);
531 //            return subject;
532 //        }
533 //    }
534 //    return Link.Create(@object, Net.And, subject);
535 //}
536
537 public static int ChunkSize = 2;
538
539 //public static Link FromList(List<Link> links)
540 //{
541 //    Link element = links[0];
542 //    for (int i = 1; i < links.Count; i += ChunkSize)
543 //    {
544 //        int j = (i + ChunkSize - 1);
545 //        j = j < links.Count ? j : (links.Count - 1);
546 //        Link subElement = links[j];
547 //        while (--j >= i) LeftJoin(ref subElement, links[j]);
548 //        RightJoin(ref element, subElement);
549 //    }
550 //    return element;
551 //}
552
553 //public static Link FromList(Link[] links)
554 //{
555 //    Link element = links[0];
556 //    for (int i = 1; i < links.Length; i += ChunkSize)
557 //    {
558 //        int j = (i + ChunkSize - 1);
559 //        j = j < links.Length ? j : (links.Length - 1);
560 //        Link subElement = links[j];
561 //        while (--j >= i) LeftJoin(ref subElement, links[j]);
562 //        RightJoin(ref element, subElement);
563 //    }
564 //    return element;
565 //}
566
567 //public static Link FromList(ICollection<Link> links)
568 //{
569 //    Link element = links[0];
570 //    for (int i = 1; i < links.Count; i += ChunkSize)
571 //    {
572 //        int j = (i + ChunkSize - 1);
573 //        j = j < links.Count ? j : (links.Count - 1);
574 //        Link subElement = links[j];
575 //        while (--j >= i)
576 //        {
577 //            Link x = links[j];
578 //            subElement = CombinedJoin(ref x, ref subElement);
579 //        }
580 //        element = CombinedJoin(ref element, ref subElement);
581 //    }
582 //    return element;
583 //}
584 //public static Link FromList(ICollection<Link> links)
585 //{
586 //    int i = 0;

```

```

587 //     Link element = links[i++];
588 //     if (links.Count % 2 == 0)
589 //     {
590 //         element = CombinedJoin(element, links[i++]);
591 //     }
592 //     for (; i < links.Count; i += 2)
593 //     {
594 //         Link doublet = CombinedJoin(links[i], links[i + 1]);
595 //         element = CombinedJoin(ref element, ref doublet);
596 //     }
597 //     return element;
598 // }
599
600 // Заглушка, возможно опасная
601 private static Link CombinedJoin(Link element, Link link)
602 {
603     return CombinedJoin(ref element, ref link);
604 }
605
606 //public static Link FromList(List<Link> links)
607 //{
608 //     int i = links.Count - 1;
609 //     Link element = links[i];
610 //     while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
611 // → links, ref i);
612 //     return element;
613 // }
614 //public static Link FromList(Link[] links)
615 //{
616 //     int i = links.Length - 1;
617 //     Link element = links[i];
618 //     while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
619 // → links, ref i);
620 //     return element;
621 // }
622 //public static Link FromList(List<Link> links)
623 //{
624 //     Link element = links[0];
625 //     for (int i = 1; i < links.Count; i += ChunkSize)
626 //     {
627 //         int j = (i + ChunkSize - 1);
628 //         j = j < links.Count ? j : (links.Count - 1);
629 //         Link subElement = links[j];
630 //         while (--j >= i) subElement = CombinedJoin(links[j], subElement);
631 //         element = CombinedJoin(element, subElement);
632 //     }
633 //     return element;
634 // }
635 //public static Link FromList(Link[] links)
636 //{
637 //     Link element = links[0];
638 //     for (int i = 1; i < links.Length; i += ChunkSize)
639 //     {
640 //         int j = (i + ChunkSize - 1);
641 //         j = j < links.Length ? j : (links.Length - 1);
642 //         Link subElement = links[j];
643 //         while (--j >= i) subElement = CombinedJoin(links[j], subElement);
644 //         element = CombinedJoin(element, subElement);
645 //     }
646 //     return element;
647 // }
648 //public static Link FromList(ICollection<Link> links)
649 //{
650 //     int leftBound = 0;
651 //     int rightBound = links.Count - 1;
652 //     if (leftBound == rightBound)
653 //     {
654 //         return links[0];
655 //     }
656 //     Link left = links[leftBound];
657 //     Link right = links[rightBound];
658 //     long leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
659 //     long rightReferers = right.ReferersBySourceCount + right.ReferersByTargetCount;
660 //     while (true)
661 //     {
662 //         //if (rightBound % 2 != leftBound % 2)
663 //         if (rightReferers >= leftReferers)
664 //         {

```

```

663         int nextRightBound = --rightBound;
664         if (nextRightBound == leftBound)
665         {
666             var x = CombinedJoin(ref left, ref right);
667             return x;
668         }
669         else
670         {
671             Link nextRight = links[nextRightBound];
672             right = CombinedJoin(ref nextRight, ref right);
673             rightReferers = right.ReferersBySourceCount +
↪ right.ReferersByTargetCount;
674         }
675     }
676     else
677     {
678         int nextLeftBound = ++leftBound;
679         if (nextLeftBound == rightBound)
680         {
681             return CombinedJoin(ref left, ref right);
682         }
683         else
684         {
685             Link nextLeft = links[nextLeftBound];
686             left = CombinedJoin(ref left, ref nextLeft);
687             leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
688         }
689     }
690 }
691 //}
692 //public static Link FromList(IList<Link> links)
693 //{
694 //    int i = links.Count - 1;
695 //    Link element = links[i];
696 //    while (--i >= 0)
697 //    {
698 //        LeftJoin(ref element, links[i]); // LeftJoin(ref element, links[i]);
699 //    }
700 //    return element;
701 //}
702
703 public static Link FromList(List<Link> links)
704 {
705     var i = links.Count - 1;
706     var element = links[i];
707     while (--i >= 0)
708     {
709         var x = links[i];
710         element = CombinedJoin(ref x, ref element); // LeftJoin(ref element, links[i]);
711     }
712     return element;
713 }
714
715 public static Link FromList(Link[] links)
716 {
717     var i = links.Length - 1;
718     var element = links[i];
719     while (--i >= 0)
720     {
721         element = CombinedJoin(ref links[i], ref element); // LeftJoin(ref element,
↪ links[i]);
722     }
723     return element;
724 }
725 }
726 }

```

1.11 ./csharp/Platform.Data.Triplets/Sequences/SequenceHelpers.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Data.Sequences;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Triplets.Sequences
9  {
10     /// <remarks>
11     /// TODO: Check that CollectMatchingSequences algorithm is working, if not throw it away.

```

```

12  /// TODO: Think of the abstraction on Sequences that can be equally usefull for triple
13  ↪ links, doublet links and so on.
14  /// </remarks>
15  public static class SequenceHelpers
16  {
17      public static readonly int MaxSequenceFormatSize = 20;
18
19      //public static void DeleteSequence(Link sequence)
20      //{
21      //}
22
23      public static string FormatSequence(Link sequence)
24      {
25          var visitedElements = 0;
26          var sb = new StringBuilder();
27          sb.Append('{');
28          StopableSequenceWalker.WalkRight(sequence, x => x.Source, x => x.Target, x =>
29              ↪ x.Linker != Net.And, element =>
30              {
31                  if (visitedElements > 0)
32                  {
33                      sb.Append(',');
34                  }
35                  sb.Append(element.ToString());
36                  visitedElements++;
37                  if (visitedElements < MaxSequenceFormatSize)
38                  {
39                      return true;
40                  }
41                  else
42                  {
43                      sb.Append(", ...");
44                      return false;
45                  }
46              });
47          sb.Append('}');
48          return sb.ToString();
49      }
50
51      public static List<Link> CollectMatchingSequences(Link[] links)
52      {
53          if (links.Length == 1)
54          {
55              throw new InvalidOperationException("Подпоследовательности с одним элементом не
56              ↪ поддерживаются.");
57          }
58          var leftBound = 0;
59          var rightBound = links.Length - 1;
60          var left = links[leftBound++];
61          var right = links[rightBound--];
62          var results = new List<Link>();
63          CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
64          return results;
65      }
66
67      private static void CollectMatchingSequences(Link leftLink, int leftBound, Link[]
68      ↪ middleLinks, Link rightLink, int rightBound, ref List<Link> results)
69      {
70          var leftLinkTotalReferers = leftLink.ReferersBySourceCount +
71          ↪ leftLink.ReferersByTargetCount;
72          var rightLinkTotalReferers = rightLink.ReferersBySourceCount +
73          ↪ rightLink.ReferersByTargetCount;
74          if (leftLinkTotalReferers <= rightLinkTotalReferers)
75          {
76              var nextLeftLink = middleLinks[leftBound];
77              var elements = GetRightElements(leftLink, nextLeftLink);
78              if (leftBound <= rightBound)
79              {
80                  for (var i = elements.Length - 1; i >= 0; i--)
81                  {
82                      var element = elements[i];
83                      if (element != null)
84                      {
85                          CollectMatchingSequences(element, leftBound + 1, middleLinks,
86                          ↪ rightLink, rightBound, ref results);
87                      }
88                  }
89              }
90          }
91          else

```

```

84     {
85         for (var i = elements.Length - 1; i >= 0; i--)
86         {
87             var element = elements[i];
88             if (element != null)
89             {
90                 results.Add(element);
91             }
92         }
93     }
94 }
95 else
96 {
97     var nextRightLink = middleLinks[rightBound];
98     var elements = GetLeftElements(rightLink, nextRightLink);
99     if (leftBound <= rightBound)
100     {
101         for (var i = elements.Length - 1; i >= 0; i--)
102         {
103             var element = elements[i];
104             if (element != null)
105             {
106                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
107                     ↪ elements[i], rightBound - 1, ref results);
108             }
109         }
110     }
111     else
112     {
113         for (var i = elements.Length - 1; i >= 0; i--)
114         {
115             var element = elements[i];
116             if (element != null)
117             {
118                 results.Add(element);
119             }
120         }
121     }
122 }
123
124 public static Link[] GetRightElements(Link startLink, Link rightLink)
125 {
126     var result = new Link[4];
127     TryStepRight(startLink, rightLink, result, 0);
128     startLink.WalkThroughReferersAsTarget(couple =>
129     {
130         if (couple.Linker == Net.And)
131         {
132             if (TryStepRight(couple, rightLink, result, 2))
133             {
134                 return Link.Stop;
135             }
136         }
137         return Link.Continue;
138     });
139     return result;
140 }
141
142 public static bool TryStepRight(Link startLink, Link rightLink, Link[] result, int
143     ↪ offset)
144 {
145     var added = 0;
146     startLink.WalkThroughReferersAsSource(couple =>
147     {
148         if (couple.Linker == Net.And)
149         {
150             var coupleTarget = couple.Target;
151             if (coupleTarget == rightLink)
152             {
153                 result[offset] = couple;
154                 if (++added == 2)
155                 {
156                     return Link.Stop;
157                 }
158             }
159             else if (coupleTarget.Linker == Net.And && coupleTarget.Source ==
160                 ↪ rightLink)

```

```

159         {
160             result[offset + 1] = couple;
161             if (++added == 2)
162             {
163                 return Link.Stop;
164             }
165         }
166     }
167     return Link.Continue;
168 });
169 return added > 0;
170 }
171
172 public static Link[] GetLeftElements(Link startLink, Link leftLink)
173 {
174     var result = new Link[4];
175     TryStepLeft(startLink, leftLink, result, 0);
176     startLink.WalkThroughReferersAsSource(couple =>
177     {
178         if (couple.Linker == Net.And)
179         {
180             if (TryStepLeft(couple, leftLink, result, 2))
181             {
182                 return Link.Stop;
183             }
184         }
185         return Link.Continue;
186     });
187     return result;
188 }
189
190 public static bool TryStepLeft(Link startLink, Link leftLink, Link[] result, int offset)
191 {
192     var added = 0;
193     startLink.WalkThroughReferersAsTarget(couple =>
194     {
195         if (couple.Linker == Net.And)
196         {
197             var coupleSource = couple.Source;
198             if (coupleSource == leftLink)
199             {
200                 result[offset] = couple;
201                 if (++added == 2)
202                 {
203                     return Link.Stop;
204                 }
205             }
206             else if (coupleSource.Linker == Net.And && coupleSource.Target ==
207                 ↪ leftLink)
208             {
209                 result[offset + 1] = couple;
210                 if (++added == 2)
211                 {
212                     return Link.Stop;
213                 }
214             }
215             return Link.Continue;
216         });
217     return added > 0;
218 }
219 }
220 }

```

1.12 ./csharp/Platform.Data.Triplets.Tests/LinkTests.cs

```

1 using System.IO;
2 using Xunit;
3 using Platform.Random;
4 using Platform.Ranges;
5
6 namespace Platform.Data.Triplets.Tests
7 {
8     public static class LinkTests
9     {
10         public static object Lock = new object(); //-V3090
11
12         private static ulong _thingVisitorCounter;
13         private static ulong _isAVisitorCounter;
14         private static ulong _linkVisitorCounter;
15

```

```

16 static void ThingVisitor(Link linkIndex)
17 {
18     _thingVisitorCounter += linkIndex;
19 }
20
21 static void IsAVisitor(Link linkIndex)
22 {
23     _isAVisitorCounter += linkIndex;
24 }
25
26 static void LinkVisitor(Link linkIndex)
27 {
28     _linkVisitorCounter += linkIndex;
29 }
30
31 [Fact]
32 public static void CreateDeleteLinkTest()
33 {
34     lock (Lock)
35     {
36         string filename = "db.links";
37
38         File.Delete(filename);
39
40         Link.StartMemoryManager(filename);
41
42         Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
43
44         Link.Delete(link1);
45
46         Link.StopMemoryManager();
47
48         File.Delete(filename);
49     }
50 }
51
52 [Fact]
53 public static void DeepCreateUpdateDeleteLinkTest()
54 {
55     lock (Lock)
56     {
57         string filename = "db.links";
58
59         File.Delete(filename);
60
61         Link.StartMemoryManager(filename);
62
63         Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
64         Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
65         Link link = Link.Create(Link.Itself, isA, Link.Itself);
66         Link thing = Link.Create(Link.Itself, isNotA, link);
67
68         //Assert::IsTrue(GetLinksCount() == 4);
69
70         Assert.Equal(isA, isA.Target);
71
72         isA = Link.Update(isA, isA, isA, link); // Произведено замыкание
73
74         Assert.Equal(link, isA.Target);
75
76         Link.Delete(isA); // Одна эта операция удалит все 4 связи
77
78         //Assert::IsTrue(GetLinksCount() == 0);
79
80         Link.StopMemoryManager();
81
82         File.Delete(filename);
83     }
84 }
85
86 [Fact]
87 public static void LinkReferersWalkTest()
88 {
89     lock (Lock)
90     {
91         string filename = "db.links";
92
93         File.Delete(filename);
94
95         Link.StartMemoryManager(filename);

```

```

96
97     Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
98     Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
99     Link link = Link.Create(Link.Itself, isA, Link.Itself);
100    Link thing = Link.Create(Link.Itself, isNotA, link);
101    isA = Link.Update(isA, isA, isA, link);
102
103    Assert.Equal(1, thing.ReferersBySourceCount);
104    Assert.Equal(2, isA.ReferersByLinkerCount);
105    Assert.Equal(3, link.ReferersByTargetCount);
106
107    _thingVisitorCounter = 0;
108    _isAVisitorCounter = 0;
109    _linkVisitorCounter = 0;
110
111    thing.WalkThroughReferersAsSource(ThingVisitor);
112    isA.WalkThroughReferersAsLinker(IsAVisitor);
113    link.WalkThroughReferersAsTarget(LinkVisitor);
114
115    Assert.Equal(4UL, _thingVisitorCounter);
116    Assert.Equal(1UL + 3UL, _isAVisitorCounter);
117    Assert.Equal(1UL + 3UL + 4UL, _linkVisitorCounter);
118
119    Link.StopMemoryManager();
120
121    File.Delete(filename);
122    }
123 }
124
125 [Fact]
126 public static void MultipleRandomCreationsAndDeletionsTest()
127 {
128     lock (Lock)
129     {
130         string filename = "db.links";
131
132         File.Delete(filename);
133
134         Link.StartMemoryManager(filename);
135
136         TestMultipleRandomCreationsAndDeletions(2000);
137
138         Link.StopMemoryManager();
139
140         File.Delete(filename);
141     }
142 }
143
144 private static void TestMultipleRandomCreationsAndDeletions(int
145     ↪ maximumOperationsPerCycle)
146 {
147     var and = Link.Create(Link.Itself, Link.Itself, Link.Itself);
148     //var comparer = Comparer<TLink>.Default;
149     for (var N = 1; N < maximumOperationsPerCycle; N++)
150     {
151         var random = new System.Random(N);
152         var linksCount = 1;
153         for (var i = 0; i < N; i++)
154         {
155             var createPoint = random.NextBoolean();
156             if (linksCount > 2 && createPoint)
157             {
158                 var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
159                 Link source = random.NextUInt64(linksAddressRange);
160                 Link target = random.NextUInt64(linksAddressRange); //-V3086
161                 var resultLink = Link.Create(source, and, target);
162                 if (resultLink > linksCount)
163                 {
164                     linksCount++;
165                 }
166             }
167             else
168             {
169                 Link.Create(Link.Itself, Link.Itself, Link.Itself);
170                 linksCount++;
171             }
172         }
173     }

```



```

174         Link link = i + 2;
175         if (link.Linker != null)
176         {
177             Link.Delete(link);
178             linksCount--;
179         }
180     }
181 }
182 }
183 }
184 }

```

1.13 ./csharp/Platform.Data.Triplets.Tests/PersistentMemoryManagerTests.cs

```

1  using System.IO;
2  using Xunit;
3
4  namespace Platform.Data.Triplets.Tests
5  {
6      public static class PersistentMemoryManagerTests
7      {
8          [Fact]
9          public static void FileMappingTest()
10         {
11             lock (LinkTests.Lock)
12             {
13                 string filename = "db.links";
14
15                 File.Delete(filename);
16
17                 Link.StartMemoryManager(filename);
18
19                 Link.StopMemoryManager();
20
21                 File.Delete(filename);
22             }
23         }
24
25         [Fact]
26         public static void AllocateAndFreeLinkTest()
27         {
28             lock (LinkTests.Lock)
29             {
30                 string filename = "db.links";
31
32                 File.Delete(filename);
33
34                 Link.StartMemoryManager(filename);
35
36                 Link link = Link.Create(Link.Itself, Link.Itself, Link.Itself);
37
38                 Link.Delete(link);
39
40                 Link.StopMemoryManager();
41
42                 File.Delete(filename);
43             }
44         }
45
46         [Fact]
47         public static void AttachToUnusedLinkTest()
48         {
49             lock (LinkTests.Lock)
50             {
51                 string filename = "db.links";
52
53                 File.Delete(filename);
54
55                 Link.StartMemoryManager(filename);
56
57                 Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
58                 Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
59
60                 Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed
61
62                 Link.StopMemoryManager();
63
64                 File.Delete(filename);
65             }
66         }
67     }

```

```

68 [Fact]
69 public static void DetachToUnusedLinkTest()
70 {
71     lock (LinkTests.Lock)
72     {
73         string filename = "db.links";
74
75         File.Delete(filename);
76
77         Link.StartMemoryManager(filename);
78
79         Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
80         Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
81
82         Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed
83         Link.Delete(link2); // Removes both links, all "Attached" links forced to be
84                             ↪ "Detached" here
85
86         Link.StopMemoryManager();
87
88         File.Delete(filename);
89     }
90 }
91
92 [Fact]
93 public static void GetSetMappedLinkTest()
94 {
95     lock (LinkTests.Lock)
96     {
97         string filename = "db.links";
98
99         File.Delete(filename);
100
101         Link.StartMemoryManager(filename);
102
103         var mapped = Link.GetMappedOrDefault(0);
104         var mappingSet = Link.TrySetMapped(mapped, 0);
105
106         Assert.True(mappingSet);
107
108         Link.StopMemoryManager();
109
110         File.Delete(filename);
111     }
112 }
113 }
114 }

```

Index

./csharp/Platform.Data.Triplets.Tests/LinkTests.cs, 38
./csharp/Platform.Data.Triplets.Tests/PersistentMemoryManagerTests.cs, 41
./csharp/Platform.Data.Triplets/CharacterHelpers.cs, 1
./csharp/Platform.Data.Triplets/GexfExporter.cs, 4
./csharp/Platform.Data.Triplets/ILink.cs, 5
./csharp/Platform.Data.Triplets/Link.Debug.cs, 6
./csharp/Platform.Data.Triplets/Link.cs, 7
./csharp/Platform.Data.Triplets/LinkConverter.cs, 16
./csharp/Platform.Data.Triplets/LinkExtensions.cs, 18
./csharp/Platform.Data.Triplets/Net.cs, 21
./csharp/Platform.Data.Triplets/NumberHelpers.cs, 24
./csharp/Platform.Data.Triplets/Sequences/CompressionExperiments.cs, 25
./csharp/Platform.Data.Triplets/Sequences/SequenceHelpers.cs, 35