Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási nyelvek és Fordítóprogramok Tanszék

Modern C++ gráf könyvtár

Témavezető:
Porkoláb Zoltán
egyetemi docens, habil. PhD

Szerző:
Schaum Béla
programtervező informatikus BSc

Budapest, 2023

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:
Név: Schaum Béla
Neptun kód: EQBVGX

Képzési adatok:
Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)
Tagozat : Nappali

Belső témavezetővel rendelkezem

Témavezető neve: *Porkoláb Zoltán*
munkahelyének neve, tanszéke: ***ELTE IK, Programozási nyelvek és Fordítóprogramok Tanszék***
munkahelyének címe: ***1117, Budapest, Pázmány Péter sétány 1/C.***
beosztás és iskolai végzettsége: ***egyetemi docens, habil. PhD***

A szakdolgozat címe: Modern C++ gráf könyvtár

A szakdolgozat témája:
(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

Szakdolgozatomban egy graph_traits template könyvtárat hozok létre, amely a C++ nyelvi elemeivel fedi el a gráf reprezentációt, hogy a gráfalgoritmusok tetszőleges objektumon tudjanak működni. Ahhoz hogy ez megvalósuljon, a gráfot ábrázoló objektum típusát template metaprogramozással kell kielemezni, és kinyerni belőle az információkat.
Az elsődleges cél hogy a kapott típusról kiderítsük, hogy milyen ábrázolásmódot használ a csúcsok és/vagy élek tárolására. Itt a szakirodalmak 3 féle fő ábrázolásról írnak: szomszédsági lista, szomszédsági mátrix illetve éllistás tárolás. Ezen felül az opcionálisan létező a gráfhoz, a csúcsokhoz illetve az élekhez rendelt súlyt/tulajdonságot is felismerjük.
Ehhez első körben egy típus-osztályozót kell megvalósítani, ami (akár csak egy előre deklarált) osztályra is pontosan meg tudja mondani hogy milyen besorolású. (index, logikai típusú, opcionális érték, tuple, bitset, szöveg, könyvtár, konténer, fordítás időben ismert nagyságú konténer)
A cél eléréséhez ezt felhasználva rekurzívan a kapott típusra, előre definiált struktúrális felismeréssel kinyerhető a gráf reprezentáció fajtája és beépített tulajdonságai. Például ha a típus egy konténer, és ebben van egy konténer ami index típusokat tartalmaz, az egy szomszédsági listás ábrázolás. Az ilyen, előre leírt szabályok alapján felismert jellemzőket teszi elérhetővé, illetve ha nem ismeri fel a típust, akkor értelmes fordítási hibával segíti a programozót a probléma feloldásában.
Másodlagos célként a felismert jellemzőket egységesített formátumra hozza, és például egy gráf-élbejárást mindegyik reprezentációban a legoptimálisabb algoritmussá alakítja, hogy azt az algoritmusok könnyen felhasználhassák. Ha a típus fordítási időben rögzített méretű, akkor a gráf-algoritmusokban is felhasználható ez az információ, és így akár fordítás időben is lehet algoritmusokat futtatni.

Budapest, 2022. 11. 21.

Tartalomjegyzék

1. fejezet

Bevezetés

Az informatikában használt gráfok olyan eszközök, amelyekkel adatokat és azok közötti kapcsolatokat tudunk megjeleníteni csúcsok és élek segítségével. A gráfokat számos területen alkalmazzák, többek között a közlekedési hálózatoknál, az emberi kapcsolatoknál vagy a molekuláris szerkezeteknél.

1.1. A probléma

A gráfok kezelésére számos könyvtárcsomag áll rendelkezésre C++ nyelven, de sok esetben ezek a könyvtárak hiányosak, nehezen kezelhetőek, vagy túlságosan területspecifikusak.

Az összegyűjtött tapasztalataim alapján általánosságban a következőek mondhatóak el a C++ gráf könyvtárakról:

- Nehézkes a fejlesztés velük: kezdeti nagyobb energiabefektetésre van szükség, nem intuitív, időigényesebb dokumentáció olvasásra van szükség az induláshoz, illetve akár egy egyszerűbb algoritmus futtatásához is.
- Nem rugalmas a gráf ábrázolásmódja, tárolásmódja, vagy csak bonyolult sablonokkal oldható meg.
- Vagy standardizált (szöveges) gráf formátumból, vagy manuálisan lehet felépíteni a gráfstruktúrát, és nem C++ Standard Libraryhoz (STL [1]) illeszkedően.
- Ha egy fejlesztői hibát vétünk a könyvtár használata közben, akkor nem egyértelmű hibajelzéseket kapunk, nehéz megtalálni a hiba forrását, vele együtt a megoldást is.

- Nagyon általános könyvtárak nem tudnak optimális performanciát adni bizonyos szakterületi feladatokhoz.
- Nem használják fel a C++ fordítási idejében ismert adatokat, emiatt ezzel nem tudják optimalizálni a kódot. [2]
- Embedded környezetben a dinamikus memóriaallokációk miatt csak bonyolultan és erős megkötésekkel lehet használni.
- Elavult C++ szabványokra épül, így az újabb nyelvi elemekkel (lambda, coroutine [3]) nem használható.

Ezen problémákra koncentrálva kezdtem el nyílt forráskódú gráf könyvtárat fejleszteni, amely az összes felsorolt pontra ad egyfajta megoldást:

- Könnyű belépési küszöböt biztosít, hogy csak az elméleti tudással egyszerűen használható legyen.
- A tárolásmód egy absztrakciós réteg, amelynek a felismerése és konfigurációja automatikus.
- Meglévő C++-os struktúrákra is alkalmazható, bármilyen fejlesztés nélkül.
- Kihasználva a fordító hibaüzenet generálását, és ismerve tipikus programozói hibákat, specifikus üzeneteket generál.
- Ahogy az STL-ben megszokhattuk például a rendezés esetében, a függvény más-más bemeneti paramétereknél más algoritmust futtat, hogy minél optimálisabb legyen az adott inputra.
- Külön figyelem van szentelve arra, hogy allokáció-tudatosak legyenek az algoritmusok, hogy így beágyazott (embedded) környezetben is használható legyen.
- Fel van készítve a későbbi C++ szabvány implementációkra, kezdve a C++20 és C++23-ban várható fejlesztésekre.
- Nem függ más könyvtáraktól, akár önmagában is használható.

1.2. Motiváció

Multinacionális cégeknél rendszeresen visszatérő probléma, ha egy ismert és gyakran használt algoritmust kell használni, hogyan álljanak neki a megvalósításnak. Írjanak-e egy új kódot a több már meglévő hasonló problémára készült mellé, vagy használjanak egyet közülük, és ha igen melyiket írják át általánosabbra. Egy másik lehetőség, hogy egy új külsős könyvtárat integráljanak, amely kizárólag az aktuális

speciális feladatra jó, vagy épphogy annyira általános, hogy nehezen továbbfejleszthető és módosítható az emiatt készült kód a későbbiekben. A gráfalgoritmusokkal hasonló probléma jelentkezik, prototipizálás, versenyfeladat megoldás, vagy csak egyszerűen munkakörnyezeti használat esetén, főleg ha már egy kész struktúrára kell alkalmazni. Jelenleg nincs olyan módszer, ami egy függvényhívással megoldja a feladatot.

Ezt a hiányt szeretném orvosolni egy új C++ könyvtár létrehozásával, ami képes az adott, STL-ből származó struktúrákat 1.1 gráfként értelmezni, kiterjesztve az STL könyvtár szabványával rendelkező adattípusokra is (például boost::containers).

```
vector<list<int>> adj_list;
map<int, map<int, edge_prop>> assoc_adj_list;
int fix_adj_list[9][10];

vector<deque<bool>> adj_mat;
vector<vector<optional<edge_prop>>> adj_prop_mat;
pair<bitset<45>, graph_prop> compressed_triangular_adj_mat;

pair<int, int> edge_list[10];
forward_list<tuple<int, int, edge_prop>> edge_prop_list;
unordered_multimap<pair<string, string>, edge_prop> edge_multilist;

// ...

using namespaces bxlx::graph;

vector<node_t<decltype(g)>> sorted = ranges::topological_sort(g);
```

1.1. forráskód. Példa a gráf ábrázolásokra, mindegyiken egységesen működő topológiai rendezésre

Ehhez a felismert struktúrákat közös interfészre kell hozni, hogy a gráf terminológiai függvényekkel tudjuk a megfelelő részeit elérni.

Az interfész így a következő részekre osztható:

- Típusok, konstansok, tulajdonságok lekérdezése.
- Gráf, él és csúcs lekérések, lekérdezések.
- Eldöntő gráfalgoritmusok.
- Bonyolultabb lekérdező, vagy sorrendező gráfalgoritmusok.

1.3. Jelenlegi állapot

Ez egy nagyon komplex problémakör, ami túlmutat egy BSc-s szakdolgozat keretein. A minimális célkitűzésem az volt, hogy egy sorrendező algoritmust (topológikus rendezés), illetve egy eldöntő algoritmust (összefüggőség vizsgálat) lefuttatható legyen legalább egy nagyobb fordítóval (msvc, gcc, clang) legalább egy platformon (windows, macos, linux) és mindegyik gráf reprezentációval, de törekedve a platformfüggetlenségre. Sok probléma merült fel a template típusok felismerésével és klasszifikálásával, a fordítók template-metaprogramok fordításának különbözőségei miatt, de a unitteszteknek, illetve a github automatikus fordításának köszönhetően az inkrementális fordítással folyamatosan visszajelzést kaptam.

2. fejezet

Felhasználói dokumentáció

Ebben a fejezetben a programcsomag telepítését és használatát mutatom be.

2.1. Telepítés

Mivel ez a template könytár header-only (azaz csak include állományokból áll) és platformfüggetlen, ezért a telepítés általában egy letöltésből vagy másolásból áll. Ezt persze nem is mindig kell manuálisan megtennünk, hisz a kódhoz mellékelve van egy CMake[1] függőségkezelő segédfájl, amit az automata letöltő keretrendszerek, mint a conan[2], vcpkg[3], Buckaroo[4], hunter[5] vagy CPM[6] támogatnak. Arra figyelni kell, hogy legyen egy C++17 kompatibilis fordító előtelepítve, hisz maga a könyvtár épít ezen szabvány funkcióira, illetve ha szükséges, akkor a CMake értelmező program minél frissebb változata legyen elérhető.

2.1.1. Letöltés

A nyílt forráskódú programok előnye, hogy hozzáférhetőek és sokszorozhatóak, így például a githubban tárolt git adattárból a legfrisebb verzió bármikor másolható (2.1. forráskód). Persze a felhasználása csak az aktuális LICENCE feltételeit teljesítve történhet meg.

[1] `https://cmake.org`
[2] `https://conan.io`
[3] `https://vcpkg.io`
[4] `https://buckaroo.pm`
[5] `https://hunter.readthedocs.io`
[6] `https://github.com/cpm-cmake/CPM.cmake`

A másik lehetőség, hogy a szakdolgozati mellékletként csatolt kódot le kell tölteni, és így még gitre sincsen szükség. A harmadik lehetőség hogy a fent felsorolt automata letöltőrendszer egyikét használva az adott keretrendszer címzését figyelembe véve a már meglévő rendszerünkbe integráljuk.

```
> git clone https://github.com/schaumb/graph_traits.git
```

2.1. forráskód. A legfrissebb kód másolása újonan a gépre, a git parancs segítségével, a terminálból

2.1.2. Kódintegráció

A letöltött kódbázisban a `graph_traits/include` mappát meg kell adni a fordítónak mint keresési útvonal, és már használhatóvá is válik a könyvtárcsomag. Természetesen egy bonyolultabb rendszernél a külső programok általában nem kézzel vannak kezelve, így ha nem kézzel töltöttük le a szoftvert, akkor nagy valószínűséggel annak elérésével sem kell foglalkoznunk. Ennek feltétele hogy a CMake fájl helyesen legyen használva, és a `bxlx::graph` CMake target helyesen megtalálható legyen a függőségek között.

2.1.3. Telepítés ellenőrzése

A kód a `<bxlx/graph>` fájl includeal, a `bxlx::graph` névtérben érhető el. A sikeres telepítést követően, akár a 2.2. forráskód szerint lehet tesztelni.

```
#include <bxlx/graph>

static_assert(bxlx::graph::version >= "1.0.0");
static_assert(bxlx::graph::major_version >= 1);
static_assert(!bxlx::graph::is_graph_v<void>);

int main() {}
```

2.2. forráskód. A programcsomag működésének egy tesztelési lehetősége

Természetesen vannak automatikusan futtatható tesztek is, amivel ellenőrizni lehet, hogy a fordító tényleg az elvárt működést produkálja. Ez a 2.3. forráskód futtatásával érhető el, és részletesebben a fejlesztői dokumentáció tesztelés fejezetében térek ki rá.

```
> mkdir build
> cd build
> cmake ..
> cmake --build . --target test
```

2.3. forráskód. Tesztelés a CMake keretrendszerrel terminálból

2.2. A programcsomag használata

2.2.1. Elméleti áttekintő

Ahhoz, hogy a programcsomag részeit tudjuk használni és értelmezni, a következő gráfelméleti fogalmakkal tisztába kell lennünk. [4]

Gráf A gráf egy (N, E) páros, amely a csúcsok (N) és élek (E) halmazából áll.

Él A gráf élhalmazának egy eleme, amely két csúcshalmazbeli elemet köt össze, aminek jele: {i, j}.

Irányított él A gráf egy éle, amely egy bemeneti csúcsból (s), és egy kimeneti csúcsból (t) áll. Azaz egy rendezett pár (s, t).

Párhuzamos élek Olyan élek, melyek pontosan ugyanazokat a csúcsokat kötik össze.

Irányított gráf Olyan gráf, melynek minden éle irányított.

Szomszéd Két csúcsra akkor mondhatjuk, hogy szomszédok, ha van köztük él.

Szomszédsági lista Egy csúcshoz (s) tartozó gráfbéli összes él, amelyben szerepel (s).

Kimenő élek Egy csúcshoz (s) tartozó olyan lista, mely minden gráfbeli irányított élet tartalmaz, melynek bemeneti csúcsa (s).

Bejövő élek Egy csúcshoz (t) tartozó olyan lista, mely minden gráfbeli irányított élet tartalmaz, melynek kimeneti csúcsa (t).

Illetve egy pár implementációs alapfogalom is használva lesz a következőekben.

Ábrázolásmód A gráf hogyan van a memóriában ábrázolva, milyen konténerek felhasználásával, milyen elrendezésben.

Szomszédsági listás ábrázolás Olyan gráf ábrázolásmód, mely minden csúcshoz eltárolja az adott szomszédjait, vagy kimenő éleit.

Szomszédsági mátrixos ábrázolás Olyan gráf ábrázolás, mely minden csúcspárhoz eltárol egy igaz/hamis értéket, hogy létezik-e él közöttük.

Éllistás ábrázolás Olyan gráf ábrázolás, ahol az éleket egy rendezett párként tárolja.

A programcsomag a három felsorolt ábrázolásmódot képes jelenleg kezelni. Cél azonban az, hogy ne implementációt adjon az ábrázolásmódokhoz, hanem csak egy struktúrát írjon le, mely egy tetszőleges C++ típusra meg tudja mondani azt, hogy felismerhető-e gráf ábrázolásként, és ha igen, akkor melyikként.

2.2.2. Típusok, konstansok, tulajdonságok lekérdezése.

Először meg kell határoznunk az osztályból `G`, hogy milyen típusú, és mit tartalmaz. A felhasználó számára ezek az ismeretek egy egyszerű használat esetén nem szükségesek, a típushelyesség miatt mégis érdemes leírni, hogy később, a bonyolultabb interfészek szignatúrájánál legyen mire hivatkozni.

A felsorolásban található kódrészletek a `bxlx::graph` névtéren belüli kódhasználatra adnak példát.

A gráf felismeréséhez

Ahhoz hogy az könytár interfészét értelmezni tudjuk, a következő definíciók érhetőek el:

```
enum class representation_t {
  adjacency_list,
  adjacency_matrix,
  edge_list
};
```

Ez az enum segít abban, hogy a gráfábrázolást értelmezzük, illetve lekérdezzük. Későbbiekben akár bővíthető illeszkedési mátrixszal is.

```
template<class G, ...>
struct graph_traits {
  constexpr static representation_t representation = ...;
  // ...
};
```

A graph_traits osztályban gyűjtjük össze azokat a reprezentációfüggő információkat, melyek segítenek egységes interfészre hozni a gráfábrázolásokat. Az osztály

nem definiált, ha a G template paraméter nem felismerhető egyértelműen gráfként. A továbbiakban a Traits template argumentummal hivatkozunk rá.

```
template<class G, ...>
constexpr bool is_graph_v = ...; // (1)

template<class G, ...>
constexpr bool it_is_a_graph_v = ...; // (2)
```

Kétféleképpen lehet lekérdezni a könyvtártól, hogy felismerte-e a template argumentumként megadott osztályt gráfként. Megkülönböztetjük a SFINAE [5] friendly (1), illetve a fordítási időben hibát adó (2) verziókat. Igaz az értéke, ha a Traits definiált, és létezik benne representation adattag. A fordítási időben hibát adó verzió kielemzi a típust, és a kiszámított információkból egy olyan fordítási hibaüzenetet ad (static_assert), amely segít abban, hogy miért nem sikerült felismerni a típust. A hibaüzenetek fejezetben részletesebben érintjük a témát.

```
template<class G, class Traits = graph_traits<G>>
constexpr representation_t representation_v = Traits::
    representation;
```

Ez a fordítás idejű változó segít, hogy egyszerűbben lekérdezhető legyen a felismert reprezentáció. Ha a gráf nem ismerhető fel, fordítási hiba keletkezik.

Példák

```
using graph_1 = vector<list<int>>;
using graph_2 = bitset<36>;
using graph_3 = forward_list<tuple<string, string>>;
using n_graph = vector<float>;

ASSERT( is_graph_v<graph_1>);
ASSERT( is_graph_v<graph_2>);
ASSERT( is_graph_v<graph_3>);
ASSERT(!is_graph_v<n_graph>);

ASSERT(representation_v<graph_1> == adjacency_list);
ASSERT(representation_v<graph_2> == adjacency_matrix);
ASSERT(representation_v<graph_3> == edge_list);

ASSERT(!it_is_a_graph_v<n_graph>); // compile error inside template
    instantiation
```

Konstansok lekérdezése

```
template<class G, class Traits = graph_traits<G>>
constexpr std::size_t max_node_size_v = ...;

template<class G, class Traits = graph_traits<G>>
constexpr std::size_t max_edge_size_v = ...;
```

Ha fordítási időben ismert a maximálisan eltárolható, vagy eltárolt csúcsok, élek száma, akkor azoknak a konstans értékét adja meg, ha nem, akkor `numeric_limits<size_t>::max()` -al tér vissza.

```
template<class G, class Traits = graph_traits<G>>
constexpr node_t<G, Traits> invalid_node_v = ...;
```

Egy fix méretű szekvenciális szomszédsági listás adattároló képes eltárolni olyan csúcsokat, amelyeket figyelmen kívül kell hagyni az algoritmusok futtatása során, mert nem létező csúcsok. Egy fix méretű szekvenciális éllistás tárolóban is előfordulhatnak ilyen csúcsok. Ez általában egy extremális elemet jelent. Általában: `~node_t<G>{}` vagy `node_t<G>{}` ha nincs tilde operátora.

Az algoritmusok elvárják azt, hogyha vannak halott csúcsok/élek a fix méretű tárolókban, akkor azok mind a tárolók végében helyezkedjenek el.

Ha az adott gráfábrázolásnak nincs ilyen értéke, akkor fordítási hiba keletkezik.

Példák

```
using graph_1 = int[10][9];
ASSERT(max_node_size_v<graph_1> == 10);
ASSERT(max_edge_size_v<graph_1> == 90);
ASSERT(invalid_node_v<graph_1> == ~int{});

using graph_2 = array<pair<string_view, string_view>, 10>;
ASSERT(max_node_size_v<graph_2> == 20);
ASSERT(max_edge_size_v<graph_2> == 10);
ASSERT(invalid_node_v<graph_2> == string_view{});

using graph_3 = bitset<36>;
ASSERT(max_node_size_v<graph_3> == 6);
ASSERT(max_edge_size_v<graph_3> == 36);

using graph_4 = array<multimap<uint8_t, float>, 100>;
```

```
ASSERT(max_node_size_v<graph_4> == 100);
ASSERT(max_edge_size_v<graph_4> == (size_t)(-1));

using graph_5 = flat_map<string_view, flat_set<string_view>>;
ASSERT(max_node_size_v<graph_5> == (size_t)(-1));
ASSERT(max_edge_size_v<graph_5> == (size_t)(-1));
```

Tulajdonságok lekérdezése

```
template<class G, class Traits = graph_traits<G>>
constexpr bool has_graph_property_v = ...; // (1)

template<class G, class Traits = graph_traits<G>>
constexpr bool has_node_property_v = ...; // (2)

template<class G, class Traits = graph_traits<G>>
constexpr bool has_edge_property_v = ...; // (3)
```

A fenti változók megadják, hogy létezik-e a típusban elkülönített rész, mely a gráfhoz (1), a csúcsokhoz (2) és az élekhez (3) rendelt jellemzőket tartalmazza.

```
template<class G, class Traits = graph_traits<G>>
constexpr bool is_user_defined_node_type_v = ...; // (1)

template<class G, class Traits = graph_traits<G>>
constexpr bool is_user_defined_edge_type_v = ...; // (2)
```

A jelenlegi implementáció felismeri azt, hogy a felhasználó által megadott gráfban a csúcsok és az élek azok a felhasználó által adott típusok.

Ha a csúcsok vagy az élek asszociatív (map típusú) konténerben helyezkednek el, akkor beszélhetünk felhasználó által definiált típusról.

(2) csak akkor létezik, ha edge típus létezik.

```
template<class G, class Traits = graph_traits<G>>
constexpr bool has_node_container_v = ...; // (1)

template<class G, class Traits = graph_traits<G>>
constexpr bool has_adjacency_container_v = ...; // (2)

template<class G, class Traits = graph_traits<G>>
constexpr bool has_edge_list_container_v = ...; // (3)
```

```
template<class G, class Traits = graph_traits<G>>
constexpr bool has_edge_container_v = ...; // (4)

template<class G, class Traits = graph_traits<G>>
constexpr bool has_in_adjacency_container_v = ...; // (5)
```

Ezekkel a traitekkel lehet lekérni azt, hogy az adott gráf típusban milyen konténer van definiálva.

A csúcsok tárolójában (1) általában a csúcsokhoz rendelt szomszédok, illetve tulajdonságok találhatóak.

A szomszédsági tárolóban (2) a szomszédok listája található, illetve az élekhez rendelt tulajdonságok vagy élindexek is megjelenhetnek itt.

Az éllistás tárolóban (3) az éleknek a listája opcionálisan a hozzájuk tartozó tulajdonságokkal van eltárolva.

Az éleket tartalmazó tárolóban (4) az élekhez rendelt tulajdonságok vannak eltárolva az élekhez tartozó élindexszel megcímezhetően.

A bejövő szomszédsági tárolóban (5) a bejövő szomszédok indexe, illetve az élindexek lehetnek eltárolva. Ez akár különböző is lehet a szomszédsági tárolótól.

```
template<class G, class Traits = graph_traits<G>>
constexpr bool has_in_edges_v = ...; // (1)

template<class G, class Traits = graph_traits<G>>
constexpr bool parallel_edges_v = ...; // (2)

template<class G, class Traits = graph_traits<G>>
constexpr bool directed_edges_v = ...; // (3)

template<class G, class Traits = graph_traits<G>>
constexpr bool compressed_edges_v = ...; // (4)
```

Ezek a tulajdonságok az eltárolt élekről adnak olyan információkat, melyeket ki lehet nyerni a típusból.

Az (1)-vel lekérdezhető, hogy az ábrázolásmód tudja-e, hogy az élek minden csúcsra el vannak különítve: ki, illetve bemenőekre, vagy sem.

A párhuzamos élek tulajdonság (2) nem minden esetben létezik. Csak akkor definiált, ha a típusokból egyértelműen kideríthető. Például ha egy szomszédsági listának a szomszédsági tárolója egy olyan asszociatív konténer, amely nem enged meg

kulcsegyezőséget, akkor nem lehetnek párhuzamosak az élek. Viszont ha szekvenciális konténert tartalmaz, akkor arról nem dönthető el egyértelműen. A szomszédsági mátrixban, mivel igaz-hamis értékeket, vagy opcionálisakat tárol el, ezért ott nem létezhetnek párhuzamos élek.

Az irányított élek tulajdonság (3) sem minden esetben létezik. Van amikor ez könnyen megállapítható, például egy fix méretű tömörített szomszédsági mátrix esetében (ha négyzetszám, akkor irányított, ha N*(N-1), akkor irányítatlan), de általában nem.

A tömörített élek (4) definíció csak a szomszédsági mátrix ábrázolásnál értelmezett, és visszaadja, hogy egy konténerben vannak-e eltárolva az élek létezése, vagy sem. Ez azt jelenti, hogy a csúcsoknak a kimenő élei nem külön-külön vannak tárolva, hanem az összes él egy nagy tárolóban található.

```
template<class G, class Traits = graph_traits<G>>
constexpr bool has_invalid_node_v = ...;
```

A fenti kód megmondja, hogy értelmezett-e invalid node 2.10 az adott gráfban.

Példák

```
using graph_1 = vector<vector<int>>;

ASSERT(!has_graph_property_v<graph_1>);
ASSERT(!has_node_property_v<graph_1>);
ASSERT(!has_edge_property_v<graph_1>);
ASSERT(!is_user_defined_node_type_v<graph_1>);
ASSERT(has_node_container_v<graph_1>);
ASSERT(has_adjacency_container_v<graph_1>);
ASSERT(!has_edge_list_container_v<graph_1>);
ASSERT(!has_edge_container_v<graph_1>);
ASSERT(!has_in_adjacency_container_v<graph_1>);
ASSERT(!has_in_edges_v<graph_1>);
ASSERT(!has_invalid_node_v<graph_1>);

// ASSERT(!is_user_defined_edge_type_v<graph_1>);
// ASSERT(parallel_edges_v<graph_1>);
// ASSERT(directed_edges_v<graph_1>);
// ASSERT(compressed_edges_v<graph_1>);
// parallel_edges_v, directed_edges_v es compressed_edges_v nincs
   ertelmezve graph_1 -en, igy ezek forditasi hibat okoznak
```

```
// tovabbiakban csak az igaz allitasokat sorolom fel (vagy a
   letezesuket ahol van ertelme), a tobbi vagy nem letezik, vagy
   hamis.

using graph_2 = pair<vector<optional<edge_prop>>, graph_prop>;
ASSERT(has_graph_property_v<graph_2>);
ASSERT(has_edge_property_v<graph_2>);
ASSERT(has_adjacency_container_v<graph_2>);
ASSERT(!parallel_edges_v<graph_2>);
ASSERT(compressed_edges_v<graph_2>);

using graph_3 = pair<map<string_view, node_prop>,
  multimap<pair<string_view, string_view>, edge_prop>>;
ASSERT(has_node_property_v<graph_3>);
ASSERT(has_edge_property_v<graph_3>);
ASSERT(is_user_defined_node_type_v<graph_3>);
ASSERT(has_node_container_v<graph_3>);
ASSERT(has_edge_list_container_v<graph_3>);
ASSERT(parallel_edges_v<graph_3>);
ASSERT(directed_edges_v<graph_3>);

using graph_4 = pair<vector<pair<list<int>,
  map<int, edge>>>, map<edge, edge_prop>>;
ASSERT(has_edge_property_v<graph_4>);
ASSERT(is_user_defined_edge_type_v<graph_4>);
ASSERT(has_node_container_v<graph_4>);
ASSERT(has_adjacency_container_v<graph_4>);
ASSERT(has_edge_container_v<graph_4>);
ASSERT(has_in_adjacency_container_v<graph_4>);
ASSERT(has_in_edges_v<graph_4>);
ASSERT(!parallel_edges_v<graph_4>);
ASSERT(directed_edges_v<graph_4>);

using graph_5 = vector<pair<size_t, deque<short>>>;
// using graph_5 = vector<pair<list<short>::iterator,
//                             list<short>>>;
ASSERT(has_node_container_v<graph_5>);
ASSERT(has_adjacency_container_v<graph_5>);
ASSERT(has_in_edges_v<graph_5>);
ASSERT(directed_edges_v<graph_5>);
```

Típusok lekérdezése

```
template<class G, class Traits = graph_traits<G>>
using graph_property_t = ...; // (1)

template<class G, class Traits = graph_traits<G>>
using node_property_t = ...; // (2)

template<class G, class Traits = graph_traits<G>>
using edge_property_t = ...; // (3)
```

Ha léteznek a gráfban elérhető gráf (1)/csúcs (2)/él (3) tulajdonságok, akkor azoknak a típusát adja vissza.

```
template<class G, class Traits = graph_traits<G>>
using node_t = ...; // (1)

template<class G, class Traits = graph_traits<G>>
using edge_t = ...; // (2)

template<class G, class Traits = graph_traits<G>>
using edge_repr_t = ...; // (3)
```

Visszaadják azt, hogy a gráfban a csúcsok, (1) illetve az élek (2) indexei hogyan vannak reprezentálva. Ez kétféle lehet attól függően, hogy szekvenciális vagy asszociatív konténer a csúcsok/élek tárolója. Ha szekvenciális tárolóban van tárolva, akkor valamilyen index (integral) [0..|N|) típusú; viszont ha asszociatív, akkor az ott tárolt tetszőleges típus.

A csúcsok, illetve élek indexének egyértelműnek kell lennie, a gráf sémájában csak egyféle típusként képes felismerni a könyvtár. Azaz például az éllistás ábrázolásban nem lehet a bemeneti csúcs más típusú mint a kimeneti csúcs. Ha nem derül ki a gráf típusából, mint például a szomszédsági mátrixnál, akkor alapértelmezetten a csúcstároló size_type típusa lesz.

Van amikor párhuzamos éleket kell egymástól megkülönböztetni, ha vannak, vagy nem egyértelmű, hogy vannak-e. A (3) általában egy olyan típus, ami az élek, éllista vagy szomszédsági lista tárolójának iterátora. Ilyen típus akkor is van, amikor él típus és él tároló nincs.

```
template<class G, class Traits = graph_traits<G>>
using node_container_t = ...;

template<class G, class Traits = graph_traits<G>>
using adjacency_container_t = ...;

template<class G, class Traits = graph_traits<G>>
using edge_list_container_t = ...;

template<class G, class Traits = graph_traits<G>>
using edge_container_t = ...;

template<class G, class Traits = graph_traits<G>>
using in_adjacency_container_t = ...;
```

Visszaadják a tárolókat, ha léteznek. További információkat a 2.14-nél lehet találni.

Példák

```
using graph_1 = vector<vector<int>>;
SAME(node_t<graph_1>, int);
SAME(edge_repr_t<graph_1>, vector<int>::const_iterator);
SAME(node_container_t<graph_1>, graph_1);
SAME(adjacency_container_t<graph_1>, vector<int>);

using graph_2 = pair<vector<optional<edge_prop>>, graph_prop>;
SAME(graph_property_t<graph_2>, graph_prop);
SAME(edge_property_t<graph_2>, edge_prop);
SAME(node_t<graph_2>, size_t);
SAME(edge_repr_t<graph_2>, vector<optional<edge_prop>>::
   const_iterator);
SAME(adjacency_container_t<graph_2>, vector<optional<edge_prop>>);

using graph_3 = pair<map<string_view, node_prop>,
  multimap<pair<string_view, string_view>, edge_prop>>;
SAME(node_property_t<graph_3>, node_prop);
SAME(edge_property_t<graph_3>, edge_prop);
SAME(node_t<graph_3>, string_view);
SAME(edge_t<graph_3>, pair<string_view, string_view>);
SAME(edge_repr_t<graph_3>, multimap<pair<string_view, string_view>,
     edge_prop>::const_iterator);
```

```
SAME(node_container_t<graph_3>, map<string_view, node_prop>);
SAME(edge_list_container_t<graph_3>,
     multimap<pair<string_view, string_view>, edge_prop>);

using graph_4 = pair<vector<pair<list<int>,
  map<int, edge>>>, map<edge, edge_prop>>;
SAME(edge_property_t<graph_4>, edge_prop);
SAME(node_t<graph_4>, int);
SAME(edge_t<graph_4>, edge);
SAME(edge_repr_t<graph_4>, map<edge, edge_prop>::const_iterator);
SAME(node_container_t<graph_4>,
     vector<pair<list<int>, map<int, edge>>>);
SAME(adjacency_container_t<graph_4>, map<int, edge>);
SAME(edge_container_t<graph_4>, map<edge, edge_prop>);
SAME(in_adjacency_container_t<graph_4>, list<int>);

using graph_5 = vector<pair<size_t, deque<short>>>;
SAME(node_t<graph_5>, short);
SAME(edge_repr_t<graph_5>, deque<short>::const_iterator);
SAME(node_container_t<graph_5>, graph_5);
SAME(adjacency_container_t<graph_5>, deque<short>);
```

Ha ezek megvannak, ebből ki tudjuk számolni azt, hogy hogyan kell megcímezni egy csúcsot, hogyan lehet lekérdezni egy csúcs tulajdonságát, stb. Fontos megjegyezni, hogy számít az is, hogy ismert-e fordítási időben a tárolók mérete, illetve hogy kinyerhető-e tetszőleges felhasználói csúcs típus esetén a csúcsok tárolójából, vagy a belső tárolóból az, hogy két csúcsról hogy döntjük el, hogy azok egyenlőek-e.

2.2.3. Gráf, él és csúcs lekérések, lekérdezések.

Ebben a részben már a futásidőre is hatással lévő függvényeket sorolunk fel, amelyekkel a gráf struktúrájában lévő adatokat tudjuk kinyerni. Még mindig a `bxlx::graph` névtérben vagyunk, a meghívható függvények esetén függvényszignatúrával körítve. Vannak opcionális, vagy feltételes argumentumok, amelyek az egyszerűség kedvéért szögletes zárójelek között lesznek jelezve. A függvény szignatúrákban a gráfok akár bal és jobbreferenciaként, illetve akár pointerként is megjelenhetnek, ami nincs itt jelölve. Ilyenkor a kimenet is megegyezik a bemeneti referencia és pointer tulajdonságokkal. Az egységes kezelés típus-módosítókkal van megvalósítva [5].

Lekérő műveletek

```
template<class G, class Traits = graph_traits<G>>
constexpr auto nodes(G&&)
  -> node_container_t<G, Traits> [[const] &/&&]; // (1)

template<class G, class Traits = graph_traits<G>>
constexpr auto edges(G&&)
  -> edge_container_t<G, Traits> [[const] &/&&]; // (2)

template<class G, class Traits = graph_traits<G>>
constexpr auto edge_list(G&&)
  -> edge_list_container_t<G, Traits> [[const] &/&&]; // (3)

template<class G, class Traits = graph_traits<G>>
constexpr auto adjacents(G&&, node_t<G, Traits> const&)
  -> adjacency_container_t<G, Traits> [[const] &/&&]; // (4)

template<class G, class Traits = graph_traits<G>>
constexpr auto adjacents(G*, node_t<G, Traits> const&)
  -> adjacency_container_t<G, Traits> *; // (5)

template<class G, class Traits = graph_traits<G>>
constexpr auto in_adjacents(G&&, node_t<G, Traits> const&)
  -> in_adjacency_container_t<G, Traits> [[const] &/&&]; // (6)

template<class G, class Traits = graph_traits<G>>
constexpr auto in_adjacents(G*, node_t<G, Traits> const&)
  -> in_adjacency_container_t<G, Traits> *; // (7)
```

Visszaadja a gráfban megtalálható konténereket, ha léteznek. Ha nem léteznek a visszatérési típusok, akkor a függvények sem. A paraméterként megkapott gráf constságát, illetve referenciáját megtartva adja vissza. Ez azt jelenti, hogyha például const gráfra van meghívva, akkor const tárolót ad vissza. (1) (2) (3) Konstans futásidejű algoritmusok.

(4) (6) Megnézi hogy létezik-e a csúcs. Ha nem, akkor std::out_of_range kivételt dob, vagy ha nincsenek bekapcsolva a kivételek, akkor std::abort-ot hív.

(5) (7) Megnézi hogy létezik-e a csúcs. Ha nem, akkor nullptr-el tér vissza.

```
template<class G, class Traits = graph_traits<G>>
constexpr auto invalid_edge(G const& graph)
    -> edge_repr_t<G, Traits>; // (0)

template<class G, class Traits = graph_traits<G>>
constexpr auto get_edge(G const& graph,
                        node_t<G, Traits> const& from,
                        node_t<G, Traits> const& to)
    -> edge_repr_t<G, Traits>; // (1)

template<class G, class Traits = graph_traits<G>>
constexpr auto get_edge(G const&,
                        edge_t<G, Traits> const&)
    -> edge_repr_t<G, Traits>; // (2)

template<class G, class Traits = graph_traits<G>>
constexpr auto equal_edges(G const& graph,
                           node_t<G, Traits> const& from,
                           node_t<G, Traits> const& to)
    -> pair<edge_repr_t<G, Traits>, edge_repr_t<G, Traits>>; // (3)

template<class G, class Traits = graph_traits<G>>
constexpr auto equal_edges(G const&, edge_t<G, Traits> const&)
    -> pair<edge_repr_t<G, Traits>, edge_repr_t<G, Traits>>; // (4)

template<class G, class Traits = graph_traits<G>>
constexpr auto get_adjacency(G const& graph,
                             node_t<G, Traits> const& from,
                             node_t<G, Traits> const& to)
    -> edge_repr_t<G, Traits>; // (5)
```

Csúcsindexekből, vagy él indexből visszaad élet (1) (2) (5), vagy az ekvivalens élek listáját (3) (4).

Ha bármelyik csúcs, vagy az adott él nem létezik, akkor (1) (2) (5) invalid iterátorral, azaz (0) visszatérési értékével tér vissza, míg (3) (4) a páron belül ugyanazon értékkel.

(1) - (4) Elvárja, hogy egy irányítatlan gráfban mindkét irányba meglegyen az él a szomszédsági listában.

(5) Annyiban különbözik, hogy irányítatlan gráfban, ha van rendezés a csúcsok között, akkor a kisebbtől a nagyobb felé nézi meg a létezést. Ha nincs rendezés,

mindkét irányba megnézi.

```
template<class G, class Traits = graph_traits<G>>
constexpr auto graph_property(G&& graph)
    -> graph_property_t<G, Traits> [[const] &/&&]; // (1)

template<class G, class Traits = graph_traits<G>>
constexpr auto node_property(G&& graph, node_t<G, Traits> const&)
    -> node_property_t<G, Traits> [[const] &/&&]; // (2)

template<class G, class Traits = graph_traits<G>>
constexpr auto node_property(G* graph, node_t<G, Traits> const&)
    -> node_property_t<G, Traits>*; // (3)

template<class G, class Traits = graph_traits<G>>
constexpr auto edge_property(G&& graph,
                             node_t<G, Traits> const& from,
                             node_t<G, Traits> const& to)
    -> edge_property_t<G, Traits> [[const] &/&&]; // (4)

template<class G, class Traits = graph_traits<G>>
constexpr auto edge_property(G* graph,
                             node_t<G, Traits> const& from,
                             node_t<G, Traits> const& to)
    -> edge_property_t<G, Traits>*; // (5)

template<class G, class Traits = graph_traits<G>>
constexpr auto edge_property(G&&, edge_t<G, Traits> const&)
    -> edge_property_t<G, Traits> [[const] &/&&]; // (6)

template<class G, class Traits = graph_traits<G>>
constexpr auto edge_property(G*, edge_t<G, Traits> const&)
    -> edge_property_t<G, Traits>*; // (7)

template<class G, class Traits = graph_traits<G>>
constexpr auto edge_property(G&& graph, edge_repr_t<G, Traits>)
    -> edge_property_t<G, Traits> [[const] &/&&]; // (8)
```

A függvények akkor léteznek, ha a visszatérési típusuk is létezik. Visszaadják a gráfhoz, a csúcshoz vagy az élhez tartozó tulajdonságot.

(2) (4) (6) Ha a paraméterként megadott csúcs/él létezik, akkor az első megtalált tulajdonságával visszatér. Ha nem létezik, akkor std::out_of_range kivétel, vagy std::abort hívás történik, attól függően hogy be van-e kapcsolva az exception kezelés.

(3) (5) (7) Ha a paraméterként megadott csúcs/él létezik, akkor az első megtalált tulajdonság címével visszatér. Ha nem létezik, akkor nullptr-el tér vissza.

Ha (8) invalid_edge(g) függvényhívás eredményét kapja meg, undefined behaviour történik.

(4) - (7) nem definiált, ha G irányítatlan gráf. Ilyen esetben a get_edge vagy get_adjacency visszatérését érdemes használni a (8) -as függvényhez, figyelve, hogy ne legyen invalid.

Példák

```
tuple<vector<string_view>, vector<tuple<int, int, int>>,
      vector<string_view>, string_view> graph_1 {
  {"node_0", "node_1"}, {{0, 1, 0}}, {"edge_0"}, "graph_prop"
}; // undirected because it has edge property sharing.

vector<tuple<vector<int>, map<int, double>, float>> graph_2 {
  { {}, {{1, 1.0}}, 2.0f },
  { {0}, {}, 3.0f }
};

ASSERT(&nodes(graph_1) == &std::get<0>(graph_1));
ASSERT(&edges(graph_1) == &std::get<2>(graph_1));
ASSERT(&edge_list(graph_1) == &std::get<1>(graph_1));
ASSERT(graph_property(graph_1) == "graph_prop");

ASSERT(adjacents(&graph_2, 0) != nullptr);
ASSERT(adjacents(&graph_2, 2) == nullptr);
ASSERT(&adjacents(graph_2, 0) == &std::get<1>(graph_2[0]));
THROWS(adjacents(graph_2, 2));

ASSERT(in_adjacents(&graph_2, 0) != nullptr);
ASSERT(in_adjacents(&graph_2, 2) == nullptr);
ASSERT(&in_adjacents(graph_2, 0) == &std::get<0>(graph_2[0]));
THROWS(in_adjacents(graph_2, 2));

ASSERT(node_property(graph_1, 0) == "node_0");
```

```
ASSERT(node_property(&graph_2, 0) != nullptr);
ASSERT(node_property(&graph_2, 2) == nullptr);
ASSERT(node_property(graph_2, 0) == 2.0f);
THROWS(node_property(graph_2, 2));

ASSERT(get_edge(graph_1, 0) != invalid_edge(graph_1));
ASSERT(get_adjacency(graph_1, 1, 0) == get_adjacency(graph_1, 0, 1)
   );
// get_edge(graph_1, 0, 1);
// undefined --> undirected, but not contains all edge both
   direction

ASSERT(edge_property(&graph_1, 0) != nullptr);
ASSERT(edge_property(graph_1, 0) == "edge_0");
ASSERT(edge_property(graph_1, 0, 1) == "edge_0");
ASSERT(edge_property(graph_1, get_edge(graph_1, 0)) == "edge_0");

ASSERT(get_edge(graph_2, 0, 1) != invalid_edge(graph_2));
ASSERT(get_edge(graph_2, 1, 0) == invalid_edge(graph_2));
ASSERT(get_edge(graph_2, 0, 2) == invalid_edge(graph_2));
ASSERT(get_adjacency(graph_2, 1, 0) != get_adjacency(graph_2, 0, 1)
   );
ASSERT(get_adjacency(graph_2, 1, 0) == invalid_edge(graph_2));

ASSERT(edge_property(&graph_2, 0, 1) != nullptr);
ASSERT(edge_property(&graph_2, 0, 2) == nullptr);
ASSERT(edge_property(graph_2, 0, 1) == 1.0);
THROWS(edge_property(graph_2, 0, 2));

ASSERT(edge_property(graph_2, get_edge(graph_2(0, 1))) == 1.0);
```

Lekérdező műveletek

```
template<class G, class Traits = graph_traits<G>>
constexpr bool has_node(G const&, node_t<G, Traits> const&); // (1)

template<class Cmp = std::equal_to<>,
         class G, class Traits = graph_traits<G>>
constexpr bool has_node(G const&, node_t<G, Traits> const&,
                        Cmp&& = {}); // (2)
```

```
template<class G, class Traits = graph_traits<G>>
constexpr bool has_edge(G const&, edge_t<G, Traits> const&); // (3)

template<class G, class Traits = graph_traits<G>>
constexpr bool has_edge(G const& graph,
                        node_t<G, Traits> const& from,
                        node_t<G, Traits> const& to); // (4)

template<class G, class Traits = graph_traits<G>>
constexpr bool has_adjacency(G const& graph,
                             node_t<G, Traits> const& from,
                             node_t<G, Traits> const& to); // (5)
```

Visszaadja, hogy létezik-e a paraméterként átvett csúcs, csúcspár vagy él index.

(1) (2) A csúcs létezés ellenőrzésének futásideje függ a csúcsok tárolójától. Ha az nem létezik, akkor lineárisan keres az éllistában.

A (2) csak akkor létezik, ha a gráf egyedi csúcsindexeket tárol, és nem létezik egyelőséget vizsgáló, vagy kisebb m funktor a gráfban.

(3) Csak abban az esetben létezik, ha létezik éleket tartalmazó tároló.

(4) (5) Az él létezésének eldöntése nagyban függ attól, hogy milyen típusú a gráf, és hogy van-e optimalizált keresés a csúcsokra, vagy élre. Irányítatlan élek esetén (3) csak a megadott irányt nézi meg a létezés szempontjából, míg (4) ha létezik rendezés, akkor a kisebbtől a nagyobb felé lévő irányt megnézi hogy létezik-e. Ha nem létezik rendezés, megnézi először az egyik irányba a létezést, ha nincs ott, akkor a másik irányba is.

```
template<class G, class Traits = graph_traits<G>>
constexpr size_t node_count(G const& graph); // (1)

template<class Cmp = std::equal_to<>,
         class G, class Traits = graph_traits<G>>
constexpr size_t node_count(G const& graph, Cmp&& cmp = {}); // (2)

template<class G, class Traits = graph_traits<G>>
constexpr size_t edge_count(G const& graph); // (3)

template<class G, class Traits = graph_traits<G>>
constexpr size_t adjacency_count(G const& graph); // (4)
```

Visszaadja, hogy a jelenlegi állapotban mennyi csúcs, illetve él/szomszédság található meg a gráfban.

(1) Visszaadja a csúcsok tárolójának méretét.

(2) Ha a csúcsok tárolója nem létezik, például éllistás ábrázolásnál, és std::is_invokable_v<Cmp, node_t<G, Traits>, node_t<G, Traits> > igaz, és a visszatérési érték bool-á konvertálható, akkor az éllista méretéhez képes akár négyzetes futásidejű is lehet, nem foglal tárolásra memóriát.

(3) Ha directed_edges_v létezik, azaz megállapítható, hogy irányított vagy irányítatlan élek vannak tárolva, akkor létezik ez a függvény is. Ha nem egyértelműen megállapítható az élek irányítottsága, akkor unspecified, hogy létezik-e a függvény, és ha létezik, akkor mit ad vissza. Irányítatlan gráfnál elvárás az, hogy mindkét irányba létezzen az összes él.

(4) Összeadja az összes szomszédsági listának a méretét. Éllistánál visszaadja az éllista méretét. Ez a függvény mindig létezik, és definiált a visszatérése.

```
template<class G, class Traits = graph_traits<G>>
constexpr auto out_edges(G const&, node_t<G, Traits> const&)
    -> range<pair<node_t<G, Traits>, edge_repr_t<G, Traits>>>; //
       (1)

template<class G, class Traits = graph_traits<G>>
constexpr auto in_edges(G const&, node_t<G, Traits> const&)
    -> range<pair<node_t<G, Traits>, edge_repr_t<G, Traits>>>; //
       (2)
```

Visszaadja a gráfban megtalálható csúcshoz tartozó ki/bejövő éleket egy belső reprezentációban létező, viewer konténer segítségével. Csak akkor léteznek a függvények, ha directed_edges_v létezik, és igazat ad vissza.

Ha a has_node szerint nem létezik az adott csúcs, akkor a függvények egy üres listát adnak vissza.

(2) Csak akkor létezik, ha éllistás ábrázolású, vagy ha has_in_edges_v igaz.

Példák

```
bitset<36> graph_1 {}; // compressed, directed
graph_1[1] = true; // 0 -> 1 edge
graph_1[8] = true; // 1 -> 2 edge
```

```
list<pair<string_view, string_view>> graph_2 {
  {"node_1", "node_2"},
  {"node_5", "node_2"},
  {"node_2", "node_5"},
};

ASSERT(has_node(graph_1, 0));
ASSERT(!has_node(graph_1, 6));
ASSERT(has_node(graph_2, "node_5"));
ASSERT(!has_node(graph_2, "no_node"));

ASSERT(has_edge(graph_1, 0, 1));
ASSERT(!has_edge(graph_1, 2, 1));
ASSERT(!has_edge(graph_2, "node_2", "node_1"));

// on these graph the has_adjacency is the same as has_edge

ASSERT(node_count(graph_1) == 6);
ASSERT(edge_count(graph_1) == 2);
ASSERT(adjacency_count(graph_1) == 2);

ASSERT(node_count(graph_2) == 3);
ASSERT(adjacency_count(graph_2) == 3);

// edge_count(graph_2)
// undefined --> not exact directed/undirected edges. No property/
   shared property used

ASSERT(size(out_edges(graph_1, 0)) == 1);
ASSERT(size(out_edges(graph_2, "node_2")) == 1);
ASSERT(size(in_edges(graph_2, "node_2")) == 2);

for (auto&& [node, repr] : out_edges(graph_2, "node_2")) {
  ASSERT(node == "node_5");
  ASSERT(repr != invalid_edge(graph_2));
}
```

2.2.4. Eldöntő gráfalgoritmusok

Ezek a gráfalgoritmusok egy magasabb szintű tulajdonságát adják meg a gráfnak. Jelenleg egy stabilan működő függvény létezik. Ennek megvalósulása volt az egyik

cél.

```
template<class G, class Traits = graph_traits<G>>
constexpr bool is_connected(G const&); // (1)

template<class Strongly = ...,
         class G, class Traits = graph_traits<G>>
constexpr bool is_connected(G const&, Strongly&& = {}); // (2)
```

(1) (2) Visszatérési értéke megadja hogy, hogy a gráf összefüggő-e. (1) Hívható, ha a típusból kiderül, hogy irányított-e. Ha irányított, akkor erős összefüggést vizsgál. Ha irányítatlan a gráf, elvárja, hogy mindkét irányba létezzenek az élek. (2) Ha fordítási időben nem derült ki, hogy irányított-e, futás időben dönti el, hogy irányítottként, vagy irányítatlanként kezelje a gráfot úgy, hogy mindkét irányba ellenőrzi az élek meglétét. Meg lehet neki adni a true_type{} / false_type{} / true / false értékeket, amivel definiáljuk, hogy alapértelmezetten milyen irányítással szeretnénk hogy használja az algoritmus, (azaz hogy léteznek-e mindkét irányba az élek, vagy sem), így optimalizálhatunk a kezdeti számításán. Ha irányított gráffal és false értékkel hívjuk meg, akkor gyengén összefüggőséget számol. Irányítatlan gráfnál false esetén az algoritmus nem számít arra, hogy egy él mindkét irányba megjelenik.

Példák

```
array<pair<vector<int>, string_view>, 10> graph {{
    {{1, 2, 4, 9},  "node 0"},
    {{0, 5},        "node 1"},
    {{0, 1},        "node 2"},
    {{8, 9},        "node 3"},
    {{7},           "node 4"},
    {{5, 5, 5},     "node 5"},
    {{0, 1, 2, 3},  "node 6"},
    {{3, 7, 9},     "node 7"},
    {{},            "node 8"},
    {{8},           "node 9"},
}};

ASSERT(is_connected(graph, std::false_type{}));
ASSERT(!is_connected(graph, true));

pair<vector<vector<pair<int, string_view>>>,
```

```
    map<string_view, float>> graph_2 {
  {
    {},
    {{{0, "edge_0"}}}
  }, {{"edge_0", 0.5f}}
};

// is_connected(graph_2); undefined behaviour, undirected graph,
// not exists on both direction all edge
ASSERT(!is_connected(graph_2, std::true_type{}));
ASSERT(is_connected(graph_2, false)); // checks both direction
```

2.2.5. Bonyolultabb lekérdező, sorrendező gráfalgoritmusok.

A következőekben lesznek olyan algoritmusok, melyek felhasználják azt, hogy az élek tulajdonságaira is kíváncsiak vagyunk, emiatt az alábbi enumot is definiáljuk. Ezen felül definiálunk egy olyan tuple típust, mely képes akár kevesebb típust tartalmazó osztállyá konvertálódni. Ennek minden részletébe most nem megyünk bele, ez a rugalmas kimenet miatt szükséges.

```
struct edge_types {
  enum type { tree, forward_or_cross, reverse,
              not_tree = forward_or_cross | reverse };
  using tree_t = std::integral_constant<type, tree>;
  using forward_or_cross_t =
    std::integral_constant<type, forward_or_cross>;
  using reverse_t = std::integral_constant<type, reverse>;
  using not_tree_t = std::integral_constant<type, not_tree>;
};
using edge_type = edge_types::type;


template<class G, class Traits, class ...Args>
struct edge_like_struct {
  optional<node_t<G, Traits>> parent;
  node_t<G, Traits> to;
  optional<const node_property_t<G, Traits>&>> to_property;
  optional<edge_repr_t<G, Traits>> edge_repr;
  Args ... other_members;
};
```

Ez fog segíteni abban, hogy a függvények kimeneténél már éltípus alapján szűrjük azt, hogy mire van szükségünk, illetve hogy a kimeneti iterátorba olyan struktúrákat is tudjunk kiírni, melyek csak egy szűkebb member-tartományt tartalmaznak csak.

Egy csúcsból induló algoritmusok

```
template<class Dist = size_t, class OutIt,
         class G, class Traits = ...>
constexpr OutIt depth_first_search(
  G const&, node_t<G, Traits> const& from,
  OutIt out, Dist max_dist = ~Dist()); // (1)

template<class Dist = size_t, class OutIt,
         class G, class Traits = ...>
constexpr OutIt breadth_first_search(
  G const&, node_t<G, Traits> const& from,
  OutIt out, Dist max_dist = ~Dist()); // (2)

template<class Weight = ..., class WeightRes = ...,
         class OutIt, class G, class Traits = ...>
constexpr OutIt shortest_paths(
  G const&, node_t<G, Traits> from,
  OutIt out, Weight = {},
  WeightRes max_weight = ~WeightRes()); // (3)
```

Az itt felsorolt algoritmusok elvárják, hogy egy irányítatlan gráfnak az éle mindkét irányba létezzen.

(1) (2) OutIt egy edge_like_struct<G, Traits, edge_types, optional<Dist> > struktúrát, vagy annak tetszőleges mennyiségű opcionális mezőjéből elhagyott tuple típust képes kiírni. [6]

(3) OutIt egy edge_like_struct<G, Traits, WeightRes> struktúrát, vagy annak tetszőleges mennyiségű opcionális mezőjéből elhagyott tuple típust képes kiírni. [6]

A Weight alapvetően egy olyan struktúra, amely az él tulajdonsággal tér vissza, ha az létezik a gráfban, és létezik rá default konstruálás, összeadás, bináris negálás és rendezés művelet. Ha nem létezik, vagy nem ilyen az él tulajdonsága, akkor konstans 1 a visszatérési érték. Létezés esetén WeightRes típusa megegyezik az él tulajdonsággal. A user defined Weightnek egy olyan funktornak kell lennie, amely a következőek valamelyikét teljesíti:

- Meghívható node, node property, node, node property, edge property ötössel, és visszatér egy számtípussal. Ez csak node és edge property létezése esetén működik.
- Meghívható node, node property, node, node property, négyessel, és visszatér egy számtípussal. Ez csak node property létezése esetén működik.
- Meghívható node, node, edge property hármassal, és visszatér egy számtípussal. Ez csak edge property létezése esetén működik.
- Meghívható node, node párral, és visszatér egy számtípussal.

Másképp működik az algoritmus akkor, ha WeightRes unsigned, illetve ha signed, hisz első esetben nem kell megnézni, hogy léteznek-e negatív körök a gráfban. Ha létezik negatív kör, akkor undefined az algoritmus működése. Illetve nem foglal a memóriában helyet, ha az iterátor random access.

Példák

```
array<pair<vector<int>, string_view>, 10> graph {{
  {{1, 2, 4, 9},  "node 0"},
  {{0, 5},        "node 1"},
  {{0, 1},        "node 2"},
  {{8, 9},        "node 3"},
  {{7},           "node 4"},
  {{5, 5, 5},     "node 5"},
  {{0, 1, 2, 3},  "node 6"},
  {{3, 7, 9},     "node 7"},
  {{},            "node 8"},
  {{8},           "node 9"},
}};

vector<tuple<int, edge_types::tree_t, size_t>> res;

depth_first_search(graph, 0, std::back_inserter(res), 3);
// we need only the node once on the tree, and they distance max 3

ASSERT(res == vector<tuple<int, edge_types::tree_t, size_t>>{
  {0, {}, 0},
  {1, {}, 1},
  {5, {}, 2},
  {2, {}, 1},
```

```
  {4, {}, 1},
  {7, {}, 2},
  {3, {}, 3},
  {9, {}, 3}
});
// we can see that 8 is closer from 0 than 3 distance (0->9->8),
// but 9 recognized at 7's neighbour, and no revisit happened.


array<tuple<int, int, edge_type>, 10> res_2 {};

auto to = breadth_first_search(graph, 4, begin(res_2));

ASSERT(res_2 == array<tuple<int, int, edge_type>, 10>{
  {4, 7, edge_type::tree},
  {7, 3, edge_type::tree},
  {7, 7, edge_type::reverse},
  {7, 9, edge_type::tree},
  {3, 8, edge_type::tree},
  {3, 9, edge_type::cross},
  {9, 8, edge_type::cross}
});

deque<tuple<int, std::string_view, double>> res_3;

shortest_paths(graph, 4, front_inserter(res_3), [](int n1, int n2)
   { return (n1+n2)/2.0; });

ASSERT(res_3 == deque<tuple<int, std::string_view, double>>{
  {8, "node 8", 16.0}, // {}
  {9, "node 9", 14.5}, // (8: 16)
  {3, "node 3", 10.5}, // (8: 16, 9: 14.5)
  {7, "node 7", 5.5} , // (3: 10.5, 9: 14.5)
  {4, "node 4", 0.0}   // (7: 5.5)
});
```

Sorrendező, kategorizáló algoritmusok

Ebből is csak egy megvalósított van, így következzen a nem hosszú interfész, amely a másik célt valósítja meg:

```
template<class OutIt, class G, class Traits = ...>
constexpr OutIt topological_sort(G const&, OutIt); // (1)
```

Az algoritmus kiírja az OutIt iterátorba a csúcsokat topológikus sorrendben. Ha a gráf nem DAG, akkor a kimenet undefined.

Példák

```
array<pair<vector<int>, string_view>, 10> graph {{
  {{1, 2, 4, 9},  "node 0"},
  {{5},           "node 1"},
  {{1},           "node 2"},
  {{8, 9, 2},     "node 3"},
  {{7},           "node 4"},
  {{},            "node 5"},
  {{0, 1, 2, 3},  "node 6"},
  {{3, 9},        "node 7"},
  {{},            "node 8"},
  {{8},           "node 9"},
}};

array<int, 10> output{};

ASSERT(topological_sort(graph, begin(output)) == end(output));
ASSERT(output == {6, 0, 4, 7, 3, 2, 1, 9, 8, 5}); // one possible
    output
```

2.3. Hibaüzenetek, hibakezelés

Az első és legfontosabb, hogyha a felhasználó nem gráf típust ad be valamelyik függvénynek, vagy nem jól struktúrálja az osztályait, akkor értelmes hibaüzenetet adjon a fordító neki. Jelenleg van egy részletes és hosszú fordítási hibaüzenet, ami pontosan leírja azt, hogy mely gráfábrázolások miért nem egyeznek a kapott típussal. Ennek egy egyszerűsített változatába lehet belefutni fordítási időben először, mely három kategóriába sorolható.

Nem egyező típusok Azaz egyik gráfábrázolást sem sikerült felismerni. Ehhez részletesebb elemzésre van szükség, hogy miért. Ennek a feloldásához készült egy python script, ami segít értelmezni a kapott fordítási hibaüzenetet.

Többszörös tulajdonság Ha valaki egy olyan típust ad meg, ahol a felismert típustulajdonságok nem egyeznek, pl a csúcs típusa a csúcsok tárolójában más, mint az éllistában, az ilyen hibát tud okozni.

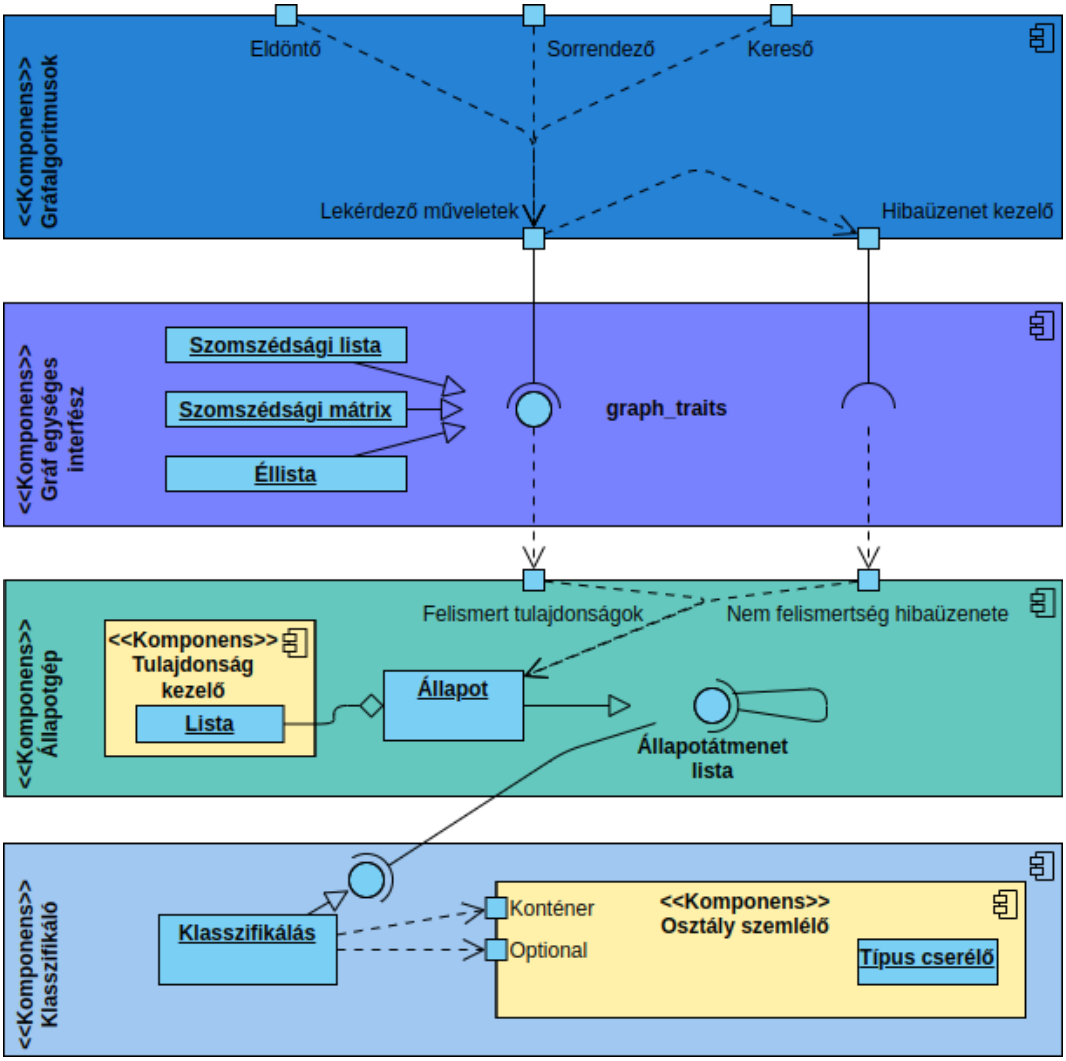
Többszörös egyezés Ritka alkalmakkor sikerül olyan struktúrát alkotni, amely többféle gráfábrázolásra is illik. Ez főleg olyankor jöhet elő, ha a gráf, csúcs vagy él tulajdonságok valamelyike egy STL konténer, és nem egy osztály.

Szerencsére a C++17-es szabványban elég sok lehetőség van arra, hogy a felhasználót már fordítási időben értesítsük [3], hogyha rossz, vagy rosszul paraméterezett függvényt akar meghívni. Ilyen esetekben fordítás idejű hibaüzenettel fog találkozni, vagy egyszerűen expliciten nem létező (=delete) függvénymegvalósításba futhat bele.

3. fejezet

Fejlesztői dokumentáció

A fejlesztés 4 fontos részből tevődött össze. Ahhoz, hogy eljussunk odáig, hogy egy tetszőleges gráf-ábrázolt típusra le tudjunk futtatni egy gráf algoritmust, először a kapott típusról ki kell derítenünk, hogy milyen ábrázolásmódot használ.



A legalsó rétegben egy típus-osztályozó (klasszifikáló), ezután egy struktúra felismerő következik (itt már kiderül az ábrázolás), és az így felismert struktúrák egy egységes interfészre (graph traits) vannak hozva. Az utolsó lépés az, hogy a tényleges gráfalgoritmusokat is implementáljuk a lekérdező műveletek segítségével.

A következő fejezetekben a rétegek bemutatása történik.

3.1. A működés

3.1.1. Típusosztályozó

A típusosztályozó egy olyan része a kódnak, mely egy osztályról fordítási időben el tudja dönteni, hogy az egy bitset, egy map, egy konténer, egy tuple, egy optional, valamilyen bool, vagy index típus. Ebben egy enum, illetve egy globális változó lesz segítségünkre.

Ha egy osztály több típushoz is tartozhat, például egy bitset lehet hogy konténer is, akkor a definíció szerint fentről lefelé az elsőhöz fog csoportosulni. Ha egyikhez se, akkor megnézzük, hogy definiált osztályról van-e szó. Ha igen, akkor indeterminate-be soroljuk, ha nem, akkor pre_declared lesz.

A következőekben menjünk végig, hogy mi alapján soroljuk a típusokat az adott osztályokhoz, mely STL-beli osztályok tartoznak ide és hogyan lehet saját típust létrehozni.

```
namespace bxlx::graph::classification {
enum class type {
  bitset,         // tomor igaz/hamis tomb
  map_like,       // kulcs-ertek parok
  range,          // tetszoleges nem map/bitset kontener
  tuple_like,     // tobb komponensu tipus
  optional,       // opcionalis ertek
  bool_t,         // logikai, vagy logikai wrapper
  index,          // indexelesre hasznalhato szam/osztaly
  indeterminate, // egyik fenti csoportba se tartozik
  pre_declared,  // nincs deklaralva
};

template<class T>
constexpr type classify = /* ... */;
}
```

Bitset

A bitset egy olyan tároló mely tömörítve tartalmaz igaz/hamis értékeket, és nem a C++ által beépített bool típusban.

Felismerése

Egy bitset típusnak nem feltétlenül kell lennie begin és end műveletének, csak annyi az elvárás, hogy létezzen operator[] függvénye, amely egy beépített bool proxy típussal tér vissza egy index típusú paramétert megadva. A bool proxynak, ami csak class lehet, kell léteznie egy noexcept bool konverziós operátorának, illetve nem szabad, hogy bool referenciával konstruálható legyen.

STL-ből példák

Ezeknek a feltételeknek az `std::bitset<N>` illetve az `std::vector<bool>` felelnek meg. [1]

Saját osztály definíció

A legkevesebb definícióval bíró ilyen osztály a következő:

```
struct my_bitset {
  struct bool_proxy {
    bool_proxy(bool&) = delete;
    operator bool() const;
  };

  bool_proxy operator[](size_t) const;
  size_t size() const;
};
```

Map típus

Ezek olyan konténerek, melyek valamilyen kulcshoz rendelnek egy vagy több értéket. Különböző stratégiájú tárolása ismert, például piros-fekete faábrázolású, vagy vödrökbe rendezett.

Felismerése

Alapból ezek a típusok konténerek is egyben, így kell léteznie begin, illetve end függvényeiknek, amely függvények egy-egy iterátort adnak vissza. Ezen felül arra van megkötés, hogy az iterátor által visszatért típusnak valamilyen tuple típusnak kell lennie, ami pontosan két komponenst tartalmaz, és léteznie kell az első komponens paraméterrel meghívható .equal_range függvényének. Sajnos ez a feltételrendszer

nem elegendő, mert bizonyos esetekben egy set-re is igaz lehet. Ezért ezen felül kell léteznie ::key_type belső típusnak, vagy .at függvényének.

STL-ből példák

A feltételeket az `std::map<K, V>`, `std::multimap<K, V>`, `std::unordered_map<K, V>`, `std::unordered_multimap<K, V>` osztályok, illetve a C++23-as szabványtól elérhető `std::flat_map<K, V>`, `std::flat_multimap<K, V>` osztályok teljesítik. [1, 7]

Saját osztály definíció

```
struct my_map {
  struct const_iterator {
    const_iterator& operator++();
    std::pair<K, V> const& operator*() const;
  };

  const_iterator begin() const;
  const_iterator end() const;
  pair<const_iterator, const_iterator> equal_range(K const&) const;

  // ezek kozul az egyik:
  using key_type = K;
  V const& at(K const&) const;
  pair<const_iterator, const_iterator> equal_range(std::pair<K, V>
      const&) const = delete;
}
```

Konténer

Annyi megkötés van ezekre a típusokra, hogy legyen egy olyan std::begin és std::end globális függvény definiálva, amelyek visszatérnek egy iterátor párral az adott konténerre. Ehhez vagy definiálni kell ezeket a std névtérbe, vagy léteznie kell az osztálynak begin, illetve end tagfüggvényének. Ide soroljuk az összes fordítási időben ismert méretű C tömböt.

STL-ből példák

Sok típus tartozik ide: `std::initializer_list<E>`, `std::array<E, N>`, `std::vector<E>`, `std::deque<E>`, `std::forward_list<E>`, `std::list<E>`, `std::set<E>`, `std::unordered_set<E>`, `std::match_results<It>`, `std::span<T>`

(C++20), `std::range::subrange<It>` (C++20), `std::flat_set<E>` (C++23), `std::flat_multiset<E>` (C++23). [1, 7]

Fontos megjegyezni, hogy bár az `std::basic_string<C>` illetve az `std::basic_string_view<C>` is ide tartozna definíció szerint, mégis ezek (illetve minden olyan osztály, amely tartalmaz length függvényt) ki van szedve a konténerek közül, hogy azok használhatóak legyenek csúcs illetve él indexként.

Saját osztály definíció

```
struct my_range {};
namespace std {
  E const* begin(my_range const&);
  E const* end(my_range const&);
}

struct my_range_2 {
  struct const_iterator {
    const_iterator& operator++();
    E const& operator*() const;
  };
  const_iterator begin() const;
  const_iterator end() const;
};
```

Tuple

Ezek olyan sablonok, melyeknek fordítási időben lekérdezhető a komponenseinek száma, és index szerint külön-külön egy-egy komponense is.

Felismerése

A felismerése egyszerű, csak megnézzük, hogy definiált-e az std::tuple_size<T> osztály, és ha igen, akkor annak value static memberje nem nulla.

STL-ből példák

Az `std::tuple<Ts...>`, `std::pair<T1, T2>` osztályok jelenleg azok, amelyek teljesítik a feltételt, és nem konténerek. [1]

Saját osztály definíció

```
struct my_tuple {};

namespace std {
  template<>
```

```
  struct tuple_size<my_tuple> : integral_constant<size_t, 2> {};
  template<>
  struct tuple_element<0, my_tuple> { using type = T1; };
  template<>
  struct tuple_element<1, my_tuple> { using type = T2; };
  template<size_t N>
  tuple_element_t<N, my_tuple> const& get(my_tuple const&);
};
```

Optional

Ez olyan sablon, mely tartalmaz egy membert, de csak feltételesen.

Felismerése

Amely osztályok bool-á konvertálhatóak, és létezik csillag operátoruk operator*, azok mind ide tartoznak. Ide soroljuk az összes nem void, nem member pointert.

STL-ből példák

Az `std::optional<T>`, illetve a nem tömböt tartalmazó smart pointerek `std::unique_ptr<T>`, `std::shared_ptr<T>` tartoznak ide. [1, 7]

Saját osztály definíció

```
struct my_optional {
  operator bool() const;
  0 const& operator*() const;
};
```

Bool

Bár ez tűnik a legegyszerűbbnek, nem csak a bool tartozik ide, hanem minden olyan típus, amely bool wrapperként viselkedik.

Felismerése

A bool típus, illetve minden olyan osztály, ami exception nélkül képes bool-ból létrejönni, illetve bool-á konvertálódni konverziós operátorral, implicit.

STL-ből példák

Az `std::atomic_bool`, ami egyedül ide sorolható.

Az `std::reference_wrapper<bool>` azért nem tartozik ide, mert nincs saját tárolója, és így nem hozható létre true illetve false konstansokból.

Saját osztály definíció

```
struct my_bool {
  my_bool(bool);
  operator bool() const;
};
```

Index

Ide sorolunk minden olyan beépített típust, illetve osztályt, amely egy szekvenciális konténer indexeléséhez használható.

Felismerése

Minden integrál típus ami nem bool és nem karakter. Ezen felül az olyan integrál wrapper típusok, melyek konvertálhatóak size_t típussá (nem bool konverziós operátorral), illetve inicializálhatóak size_t típusból implicit.

STL-ből példák

Az `std::atomic<T>`, ami egyedül ide sorolható, ha T integrál típus.

Saját osztály definíció

```
struct my_index {
  my_bool(int);
  operator int() const;
};
```

3.1.2. Struktúrafelismerő

Miután egy-egy típust külön-külön tudunk osztályozni, és nem dob hibát nekünk a fordító egy olyan osztály láttán, ami akár nem is példányosítható, illetve bele lehet nézni az osztályozott típusok altípusaiba, készen áll minden arra, hogy a struktúrát felismerjük. Ezt olyan stratégiával tesszük, hogy a rekurzív osztályozás közben egy fordítás idejű állapotgépet tartunk fenn, amely minden állapotban összegyűjt kellő információkat, minthogy asszociatív-e az adott konténer, mi a kulcs típusa, vagy hogy melyik a csúcsokhoz tartozó tulajdonság, és az hol érhető el, stb.

Az összes gráfábrázolás egy állapotgépbe van sűrítve az egyszerűség kedvéért. Ez egy olyan speciális nemdeterminisztikus állapotgép, amely egy állapot verem fenntartását is elvárja, és bizonyos állapotok közötti lépésnél nem csak egy, de akár több állapotba is vezethet, majd egy végállapotból visszatérhet a verem tetején lévő, még feldolgozatlan állapothoz.

STATE	INPUT	CONDITION	SET	NEXT STATES	NEXT INPUTS
graph	tuple N > 2	not_range(tuple[N-1])	graph_prop=tuple[N-1]	graph-p	tuple[0..N-1)
graph	tuple N = 2	not_range(tuple[1])	graph_prop=tuple[1]	graph-p	tuple[0]
graph	*	-	-	graph-p	*
graph-p	tuple N > 2	-	-	edge_container graph-ep	tuple[N-1] tuple[0..N-1)
graph-p	tuple N = 2	-	-	edge_container graph-ep	tuple[1] tuple[0]
graph-p	*	-	user_edge=NA	graph-ep	*
graph-ep	*	-	-	adj_list	*
graph-ep	*	-	-	adj_mat	*
graph-ep	*	-	-	edge_list	*
node	index	user_node=H /node_type=index/	node_type=index	-	-
node	*	user_node=I node_type=*	-	-	-
edge	index	user_edge=H /edge_type=index/	edge_type=index	-	-
edge	*	user_edge=I edge_type=*	-	-	-
edge	*	user_edge=NA undefined(edge_prop) not_range(*)	edge_prop=*	-	-
edge_container	map K V	not_range(V)	edge_type=K edge_prop=V user_edge=I	-	-
edge_container	range E	random_access(range) not_range(E)	edge_prop=E user_edge=H	-	-

3.1. táblázat. A gráf felismerő állapotgép 1/4 - Kezdő-, illetve közös állapotok

STATE	INPUT	CONDITION	SET	NEXT STATES	NEXT INPUTS
adj_list	range E	random_access(range)	user_node=H	adj_list_node	E
adj_list	map K V	not_multimap(map)	user_node=I node_type=K	adj_list_node	V
adj_list_node	tuple N > 2	not_range(tuple[N-1])	node_prop=tuple[N-1]	adj_list_node-p	tuple[0..N-1)
adj_list_node	tuple N = 2	not_range(tuple[1])	node_prop=tuple[1]	adj_list_node-p	tuple[0]
adj_list_node	*	-	-	adj_list_node-p	*
adj_list_node-p	tuple N = 2	separator_of(*tuple)	in_edges=I	adj_list_cont	tuple[1]
adj_list_node-p	tuple N = 2	-	in_edges=I	adj_list_cont adj_list_cont	tuple[0] tuple[1]
adj_list_node-p	*	-	in_edges=H	adj_list_cont	*
adj_list_cont	range E	-	-	node	E
adj_list_cont	map K V	-	-	node edge	K V

3.2. táblázat. A gráf felismerő állapotgép 2/4 - Szomszédsági listás állapotok

STATE	INPUT	CONDITION	SET	NEXT STATES	NEXT INPUTS
adj_mat	range E	random_access(range)	compressed=H	adj_mat_node	E
adj_mat	*	-	compressed=I	adj_mat_cont	*
adj_mat_node	tuple N > 2	not_range(tuple[N-1])	node_prop=tuple[N-1]	adj_mat_node-p	tuple[0..N-1)
adj_mat_node	tuple N = 2	not_range(tuple[1])	node_prop=tuple[1]	adj_mat_node-p	tuple[0]
adj_mat_node	*	-	-	adj_mat_node-p	*
adj_mat_node-p	tuple N = 2	-	in_edges=I	adj_mat_cont adj_mat_cont	tuple[0] tuple[1]
adj_mat_node-p	*	-	-	adj_mat_cont	*
adj_mat_cont	bitset	user_edge=NA /multi_edge=H/	multi_edge=H	-	-
adj_mat_cont	range E	random_access(range)	-	adj_mat_elem	E
adj_mat_elem	optional O	-	-	adj_mat_elem_opt	O
adj_mat_elem	range E	compressed=H /multi_edge=I/	multi_edge=I	edge	E
adj_mat_elem	bool	user_edge=NA /multi_edge=H/	multi_edge=H	-	-
adj_mat_elem_opt	range E	/multi_edge=I/	multi_edge=I	edge	E
adj_mat_elem_opt	*	/multi_edge=H/	multi_edge=H	edge	*

3.3. táblázat. A gráf felismerő állapotgép 3/4 - Szomszédsági mátrix állapotok

STATE	INPUT	CONDITION	SET	NEXT STATES	NEXT INPUTS
edge_list	tuple N = 2	-	-	node_container edge_list-n	tuple[0] tuple[1]
edge_list	*	-	user_node=I	edge_list-n	*
edge_list-n	range E	-	-	edge_list_elem	E
edge_list-n	map K V	-	-	edge node_pair	V K
edge_list_elem	tuple N > 2	-	-	edge node_pair	tuple[N-1] tuple[0..N-1)
edge_list_elem	tuple N = 2	-	-	edge node_pair	tuple[1] tuple[0]
edge_list_elem	*	-	-	node_pair	*
node_container	map K V	not_multimap(map) not_range(V)	node_type=K node_prop=V user_node=I	-	-
node_container	range E	random_access(range) not_range(E)	node_prop=E user_node=H	-	-
node_pair	tuple N = 2	-	-	node node	tuple[0] tuple[1]

3.4. táblázat. A gráf felismerő állapotgép 4/4 - Éllistás állapotok

Az állapotátmenet leírására szolgáló táblázatban 3.1 az első oszlop adja meg a kiinduló állapotot. Mivel nemdeterminisztikus, ezért a felsorolt bemeneti állapotokból bármelyiken tovább tud lépni. A második oszlop egy típus előszűrő, ami megmondja, hogy milyen típusra van értelmezve az átmenet. Itt van felhasználva a típusklasszifikáció, illetve a belső típus felismerése. A '*' karakternél nincs típus előszűrés. A harmadik oszlopban definiálva lehetnek specifikusabb, az eddig beállított változókra vonatkozó feltételek, és típustulajdonsági megkötések a típusklasszifikáción túl.

A táblázat 3.1 második felében az adott állapothoz tartozó továbblépési utasítások vannak. A negyedik oszlopban a beállítandó változók listáját, míg az ötödik és hatodik oszlopban a következő állapotok, illetve az azokhoz tartozó bemeneti típusokat találhatjuk. Ha valamelyik sorban nincs következő állapot, az végállapotot jelöl. Ez automatikusan nem jelenti a sikeres felismerést, csak annyit, hogy az állapotveremben tovább lehet lépni.

A felismerés folyamata úgy történik, hogy az állapotverembe beletesszük a (graph, T) párost, és addig futtatjuk az állapotgépet a verem tetején lévő állapot és típus kivételével, amíg vagy hibába nem futunk, vagy az állapotverem üres nem lesz.

A kódban az állapotgép a következőképpen érhető el:

```
using G = bxlx::graph::traits::state_machine::graph;

constexpr auto valid_v = G::template valid<T>();
```

Ekkor az állapotgép által visszaadott objektum még fordítási időben konvertálható igaz/hamis értékké, amely megadja, hogy sikerült-e felismerni a T típust gráfként. Ha sikeres volt, akkor a visszatérési érték típusában létezik egy properties altípus, ami tartalmazza az állapotgép során felismert tulajdonságokat, illetve az állapotgép állapotátmeneteinél felhasznált típusokat. Ha sikertelen volt, akkor template argumentumként tartalmazza a felismerés közbeni részletes hibát.

3.1.3. Graph traits, és az arra épülő közös interfész

Ha bármely ábrázolásmódot felismeri a struktúrafelismerő, akkor az állapotgépekben összegyűjtött információk alapján egy típusban mindezt kiajánljuk a magasabb rétegeknek úgy, hogy valamennyire már közös interfészre legyen hozva. Természetesen a közös interfész nem tartalmaz bonyolultabb logikát, csak fordítás idejű változókat, illetve a struktúrán belüli objektumok elérésének módszereit.

A felhasználói dokumentációban részletezett tulajdonságok, típuslekérdezések, illetve a gráf él és csúcs lekérő lekérdező függvények implementációi használják közvetlenül a graph_traits osztályt, és annak tagfüggvényeit, tagváltozóit. Ezeket nevezzük a gráfok közös interfészének.

3.1.4. Az algoritmusok

Mélységi bejárás

Az összefüggőség vizsgálat, illetve a topológikus rendezés is a mélységi bejárást alkalmazza, ezért ennek az algoritmusát mutatom be (algoritmus 1).

A mélységi bejáráshoz szükség van verem adatszerkezetre, hogy eltároljuk az épp bejárás alatt álló csúcsokat. Mivel minimalizálni akarjuk a memóriaallokációt a heap-en, ezért nem iteratív megvalósítást használunk, hanem a program stackjén tároljuk el a csúcsokat, amit rekurzióval tudunk elérni. Ahhoz hogy minimális legyen a függvényhívás költsége, csak a vizsgált csúcsot adjuk meg argumentumként a tagfüggvénynek. Bizonyos méret felett ez nem működne, ezért a csúcsok száma limitálva van ezzel a módszerrel.

algoritmus 1 Rekurzív mélységi bejárás [4]

```
 1: function MelysegiBejar(gráf G, csúcs n, bool elő = I)
 2:     Minden csúcshoz rendeljünk hamis értéket
 3:     function MelysegiBejarIn(csúcs n)
 4:         'n' csúcshoz rendeljünk igaz értéket
 5:         if elő then
 6:             írjuk ki 'n' csúcsot
 7:         end if
 8:         for all 'sz' szomszéd az 'n' csúcshoz 'G' szerint do
 9:             if 'sz' csúcshoz tartozó érték hamis then
10:                 MelysegiBejarIn(sz)
11:             end if
12:         end for
13:         if nem elő then
14:             írjuk ki 'n' csúcsot
15:         end if
16:     end function
17:     MelysegiBejarIn(n)
18: end function
```

Ezen felül szükségünk van egy csúcshalmazra, ami a már bejárt csúcsokat tárolja. Ha fordítási időben ismert a gráf csúcsszámának maximális mérete, akkor létrehoz

egy bitset-et, és abban tárolja el a bejárás meglétét. Ha nem ismert a maximális mérete, akkor egy bizonyos méretszám alatt (|N| <= 2^10) fix bitsettel dolgozik az algoritmus, afelett különböző stratégiákat használ. Ha esetleg nem index értékek a csúcsok, akkor szükséges lesz allokációra a hozzárendelő adatszerkezethez (map), ilyenkor a node_container allokátora van felhasználva arra, hogy az algoritmus eldöntse, hol foglaljon le területet.

Összefüggőség vizsgálat

Háromféle összefüggőség vizsgálatról beszélhetünk. Irányítatlan gráfban az összefüggőség annyiból áll, hogy egy csúcsból bármely más csúcsra el tudunk jutni élek segítségével. Irányított gráfban beszélhetünk erősen, illetve gyengén összefüggőről.

Irányítatlan gráf

Az irányítatlan gráfban lévő összefüggőség vizsgálatot egyszerűen lehet implementálni. Egy tetszőleges csúcsból kell indítani egy bejárást, és ha minden csúcsot elért, akkor összefüggő.

algoritmus 2 Összefüggőség vizsgálata irányítatlan gráfban [4]

```
1: Legyen a gráfunk G = (N, E)
2: Legyen a kezdő csúcsunk tetszőleges s ∈ N
3: function OSSZEFUGGO(gráf G, csúcs s)
4:     Legyen n := 0
5:     for all MelysegiBejar(G, s) kimenete do
6:         n := n + 1
7:     end for
8:     return n == |N|
9: end function
10: return Osszefuggo(G, s)
```

Irányított gráf, gyengén összefüggő

Az irányítatlan gráfhoz képest annyi a változtatás, hogy a mélységi bejárásnál a szomszédos csúcs lekérdezésénél nem csak a kimenő élek céljait, hanem a bejövő élek forrását is számításba vesszük, így az algoritmus parancsai megegyeznek.

algoritmus 3 Gyenge összefüggőség vizsgálata irányított gráfban [4]

1: Legyen a gráfunk G = (N, E)
2: Legyen a módosított gráfunk G' = (N, E ∪ E'), ahol E' = $\{(v, u) \mid (u, v) \in E\}$
3: Legyen a kezdő csúcsunk tetszőleges s ∈ N
4: **return** Osszefuggo(G', s)

Irányított gráf, erősen összefüggő

Ez is hasonló az irányítatlan gráf bejárásához. Először lefuttatjuk egy tetszőleges csúcsra az irányítatlan összefüggőségvizsgálatot, és ha igaz, akkor lefuttatjuk megint, csak minden élet megfordítunk.

algoritmus 4 Erős összefüggőség vizsgálata irányított gráfban [4]

1: Legyen a gráfunk G = (N, E)
2: Legyen a kezdő csúcsunk tetszőleges s ∈ N
3: **if** Osszefuggo(G, s) **then**
4: Legyen a módosított gráfunk G' = (N, E'), ahol E' = $\{(v, u) \mid (u, v) \in E\}$
5: **return** Osszefuggo(G', s)
6: **end if**
7: **return** Hamis

Topológikus rendezés

Az algoritmus csak körmentes, irányított gráfokra van értelmezve. Egy mélységi bejárást futtatva, a csúcsokat azok feldolgozása után kiíratva megkapjuk a topológikus sorrend fordítottját.

algoritmus 5 Topológikus rendezés DAG-ban [4]

1: Legyen a gráfunk G = (N, E)
2: Legyen V csúcsok verme
3: **for all** MelysegiBejar(G, s, H) kimenete 'n' **do**
4: Helyezzük 'n' csúcsot V verembe
5: **end for**
6: **return** V verem

3.2. Tesztelés

A szakdolgozatomat Test Driven Development technológiával fejlesztettem.

- Első lépésként az adott függvény deklarációját hoztam létre.

- Másodikként kisebb unit teszteseteket írtam, amelyek lefedik a feature halmazt, amit a függvény interfész enged.
- Utolsó lépésként az implementációt addig fejlesztettem, amíg az összes teszteset sikeres nem lett.

3.2.1. Unit tesztek

A négy fő komponens unit tesztesetei külön-külön fájlba lettek fejlesztve, a *test/* mappán belül.

Típusosztályozó tesztjei

A típusosztályozó teszjei különböző típusok klasszifikálásainak helyességéről szólnak, ellenőrizve az STL-ben található struktúrákat és saját megvalósításokat. Figyelnek a predeklarált osztályok belső megjelenésére (például konténerekben). Ezen felül ha vannak boost-beli típusok, és azok include fájljai, akkor azokat is teszteli.

classification.cpp	
függvény neve	**megjegyzés**
check_bitsets	Bitset ellenőrző [1]
check_maps	Map/Hozzárendelő ellenőrző [1]
check_ranges	Konténer ellenőrző [1][2][3]
check_tuples	Tuple ellenőrző
check_optionals	Opcionális ellenőrző
check_bools	Logikai ellenőrző [1]
check_indices	Index típus ellenőrző [1]
check_indeterminates_and_predeclareds	Egyéb típusok [4]

[1] Jelenleg a saját osztály definíció nem működik, ezért ki van commentezve
[2] GCC nem fordítja le a std::match_results-t, privát öröklődés miatt
[3] MSVC nem ismeri fel a tuple-t tartalmazó konténereket
[4] A clang 9.0 verziója internal compiler errort produkál bizonyos típusoknál

Állapotgép tesztjei

Az állapotgép tesztelése a 3.1 táblázatban fellelhető feltételeket, illetve a tulajdonsághalmaz beállítását ellenőrzi több típusra.

state_machine.cpp	
függvény neve	**megjegyzés**
check_input_tuple_size	Ha tuple, akkor a darabszám feltétel teljesül-e
check_random_accessness	Ellenőrzi a számmal indexelhető konténereket[5]
check_assoc_multi_property	Ellenőrzi, hogy egy map többszörösen tartalmazhatja-e ugyanazt a kulcsot
check_separator	Ellenőrzi, hogy a tuple második komponense konténer-e, és hogy az első elem egy iterátor vagy index, ami elválasztást jelöl [6]
check_empty_properties	Megnézi hogy egy üres tulajdonság-halmaznak lekérdezhető eleme-e
check_multi_properties	Megnézi, hogy több tulajdonságot tartalmazó halmaznak létezik-e bizonyos eleme, és ha igen, vissza tudja-e adni

Egységes interfész tesztjei

Ebben a fájlban a felismert gráf tulajdonságainak a tesztjei vannak implementálva. A megjegyzésben ezek a tulajdonságok vannak felsorolva.

graph_traits.cpp	
függvény neve	**tesztelt tulajdonságok**
check_adj_list_simple	csúcslista tároló, szomszédsági lista tároló
check_adj_list_unique_node	asszociatív csúcstároló, csúcs típusa

[5]MSVC bizonyos konténerekre fordítási hibát ad

check_adj_list_unique_edge	asszociatív éltároló, él típusa
check_adj_list_with_properties	gráf tulajdonság, csúcs tulajdonság, él tulajdonság
check_adj_list_parallel_edges	asszociatív multi csúcstároló
check_adj_list_empty_nodes	fix méretű szomszédsági tároló, invalid csúcs
check_adj_matrix_simple	csúcslista tároló, szomszédsági lista tároló
check_adj_matrix_compressed	szomszédsági lista tároló, csúcsok száma
check_adj_matrix_unique_edge	asszociatív éltároló, él típusa
check_adj_matrix_with_properties	gráf tulajdonság, csúcs tulajdonság, él tulajdonság
check_adj_matrix_parallel_edges	asszociatív multi szomszédsági tároló
check_edge_list_simple	éllista tároló
check_edge_list_unsortable_node	csúcs típusa
check_edge_list_unique_edge	asszociatív éltároló, él típusa
check_edge_list_with_properties	gráf tulajdonság, csúcs tulajdonság, él tulajdonság
check_edge_list_parallel_edges	asszociatív multi éllista tároló

Lekérő/Lekérdező műveletek

Ezek a tesztek fix, előre definiált, kézzel szerkesztett gráfokon futnak. Mindegyik tesztesetnél egy igaz és egy hamis teszteset fut.

graph_traits_retrieval.cpp	
függvény neve	**megjegyzés**
check_node_exists	Csúcsok létezését teszteli
check_node_exists_comp	Csúcsok létezését teszteli, állistás ábrázolásban
check_edge_exists_by_node	Élek létezését teszteli csúcspárok megadásával
check_edge_exists_by_edge_index	Élek létezését teszteli élindex segítségével
check_traits_containers	Lekérdezi a belső konténereket
check_traits_properties	Lekérdezi a gráf/csúcs/él tulajdonságokat

graph_traits_queries.cpp	
függvény neve	**megjegyzés**
check_node_indices	Lekér, és végigiterál a csúcsindexeken
check_out_edges	Lekér, és végigiterál egy csúcshoz tartozó kimenő éleken
check_in_edges	Lekér, és végigiterál egy csúcshoz tartozó bejövő éleken
check_neighbours	Lekér, és végigiterál egy csúcshoz tartozó kimenő és bejövő éleken

3.2.2. Funkcionális tesztek

A funkcionális teszteknél a tesztelendő gráfok definícióját és inicializálását egy külön fájlban tároljuk az újrahasznosítás érdekében. Ezek a gráfok mátrixszerűen az összes alábbi tesztre lefutnak. A unit tesztek mellett találhatóak meg ezek a tesztek, nem elkülönülve tőlük.

test_graphs.hpp	
létrehozó függvény neve	**megjegyzés**
empty_graph	Üres gráf, 0 csúcs, 0 él
one_node_graph	1 csúcs, 0 él
hundred_node_bin_tree_graph	100 csúcs, 99 él, irányított szomszédsági listás
hundred_node_full_graph	100 csúcs, 4950 él, irányított szomszédsági mátrixos
adj_list_random_graph	[50-100] csúcs [0.9 %] átlagos irányítatlan él
adj_mat_random_graph	300 csúcsos tömörített átlagos kimenő él
edge_list_random_graph	Nem meghatározott irányítású, véletlen generált szöveg substringjei a csúcsok

Összefüggőség vizsgálat és topológikus sorrend

connected.cpp	
függvény neve	**megjegyzés**
check_connection_default	A fentebbi gráfokon lefuttatja az algoritmust, ellenőrzi a tesztesetek referenciáival
check_connection_explicit_parameter	A fentebbi gráfokon lefuttatja az algoritmust igaz, majd hamis paraméterrel, majd ellenőrzi a tesztesetek referenciáival

topology.cpp	
függvény neve	**megjegyzés**
check_topo_function_existance	Megnézi, hogy létezik-e az adott gráfra függvény [7]
check_topo_output_iterator	Megnézi, és validálja a kijövő adatot

3.3. Használati összehasonlítás

Az általam implementált gráf könyvtár több új funkciót hoz be, ami más könyvtárakban nem található meg. Többek között az automatikus típusfelismerés, az egyedi csúcs/él index engedélyezése, illetve a fordítási időben felismert típus-invariánsok. Ezek ilyen formában nem összehasonlíthatóak más könyvtárral.

A legközelebb a Boost Graph Library [8] áll ehhez a megvalósításhoz, így a függvényekkel kapcsolatban a következő pontokban lehet a különbségeket összefoglalni.

Tulajdonság	Boost	Ez a könyvtár
Beépíthető gráf/csúcs/él tulajdonságok	Igen [8]	Igen [9]
Kötelezően meghatározott élirányítottság	Igen	Nem [10]
Rugalmas éllistás ábrázolás	Nem [11]	Igen
Bármely típusú gráf implementációra lefuttathatóak az algoritmusok	Nem	Igen
Tetszőleges sorrendű paraméterlista átadás	Igen [12]	Nem
Automatikus heap használat	Igen	Nem
Kimenet felhasználása az algoritmusokban	Nem	Igen
Összefüggőség vizsgálat eldöntő függvénnyel	Nem [13]	Igen
Visitor koncepció	Igen	Nem [14]
Exception dobás algoritmusnál	Igen	Nem

A táblázatba nincs beleszámítva, hogy saját osztályt, és saját implementációt is lehet használni a boost könyvtárral, ha definiáljuk a saját osztályunkhoz tartozó boost::graph_traits példányt. Ez a definiálás már haladóbb szintű felhasználót feltételez.

[8] Kivéve az éllistás ábrázolást

[9] A tulajdonságok tárolásának helye rugalmasabb

[10] Ugyanaz az ábrázolásmód használható többféleképpen is

[11] Csak irányított, párhuzamos éleket tartalmazó éllistás ábrázolás érhető el

[12] Az elsődleges ok, hogy miért nem egyszerű a függvények meghívása

[13] Meg kell hívni az összefüggő komponensekre alakítást, speciális argumentumokkal, és a kimenetet fel kell dolgozni

[14] Az STL-ben használt output iterátor újrahasznosítása történik

4. fejezet

Összegzés

A céljaim egy része látszik, hogy teljesült. Minimális befektetéssel mostmár egy egyszerű összefüggőség vizsgálat, vagy egy topológikus rendezés lefuttatható, és mindezt alacsony kódfüggőséggel illetve egyszerű interfésszel lehet megtenni.

A nyílt forráskódja révén várhatóan érdeklődést fog kiváltani más fejlesztők felől is, és többen tudunk részt venni a könyvtár további fejlesztésében.

4.1. Továbbfejlesztési ötletek

Természetesen ez a könyvtár még nem végleges. A fejlesztése közben elég sok ötlet merült fel bennem, hogy milyen irányba lehetne még bővíteni a kódot:

- Gráf módosító függvények implementálása, mint például fává alakítás, vagy csúcsok összevonása.
- Gráf "néző" struktúrák, amelyek az eredeti gráf egy részét tartalmazzák csak.
- Bemenet további általánosítása, hogy akár egy fájlból automatikusan felismert formátummal is működjenek az algoritmusok.
- Gráfábrázolás véletlen hozzáférésű fájlban, hogy ne töltse be a teljes adatot memóriába.
- A struktúrák további általánosítása, többféle struktúra felismerése, például a tuple típusok helyett structure binding-gal szétszedett tetszőleges osztályra működjön.

- Párhuzamos algoritmusok implementálása a std::execution::* [9] első paraméterként való átadásával.

Irodalomjegyzék

[1] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013. ISBN: 0321563840, 9780321563842.

[2] David Vandevoorde, Nicolai Josuttis és Douglas Gregor. *C++ Templates: The Complete Guide*. 2nd. Addison-Wesley Professional, 2017. ISBN: 0321714121, 9780321714121.

[3] Jacek Galowicz. *C++17 STL Cookbook: Discover the latest enhancements to functional programming and lambda expressions*. Packt Publishing, 2017. ISBN: 178712049X, 9781787120495.

[4] István Fekete és László Hunyadvári. *Algoritmusok és adatszerkezetek*. Digitális Tankönyvtár, 2015. ISBN: 9789632484565.

[5] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. 1st. O'Reilly Media, Incorporated, 2014. ISBN: 1491903996, 9781491903995.

[6] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. 3rd. Addison-Wesley Professional, 2005. ISBN: 0321334876, 9780321334879.

[7] Nicolai M. Josuttis. *C++17 - The Complete Guide: First Edition*. NicoJosuttis, 2019. ISBN: 396730017X, 9783967300178.

[8] Jeremy G. Siek, Lie-Quan Lee és Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002. ISBN: 0201729148, 9780201729146.

[9] Anthony Williams. *C++ Concurrency in Action*. 2nd. Manning Publications, 2019. ISBN: 1617294691, 9781617294693.