# ADA - Projecte:
# El problema del Viatjant de Comerç

Aquest document conté l'enunciat del projecte de l'assignatura amb totes les explicacions del seu funcionament.

## Continguts

# Introducció

L'objectiu d'aquest projecte és que apliqueu diverses tècniques algorítmiques per resoldre el problema del viatjant de comerç i que analitzeu i compareu els resultats obtinguts.

El problema del viatjant de comerç consisteix en buscar el camí més curt per visitar un conjunt de ciutats. En aquest projecte es considerarà que hi ha una ciutat de sortida i una de destinació final, deixant que la resta de ciutats a visitar puguin anar en qualsevol ordre. El problema es modelitzarà com un graf en el pla on cada vèrtex és un punt del pla que representa una ciutat i les arestes entre un vèrtex i un altre representen les carreteres. La longitud de les arestes serà la distància (euclidiana) entre els vèrtexs que uneixen. Les ciutats a visitar es representaran amb una llista de punts (vèrtexs) on la ciutat de sortida és el primer punt i la de destí l'últim. Les ciutats obligatòries a visitar no es poden repetir excepte si la primera i l'última coincideixen, que en aquest cas tindrem una ruta circular.

El projecte consistirà en implementar els algoritmes de cerca del camí mínim de Dijsktra i resoldre el problema del viatjant utilitzant algoritmes greedy, backtracking i branch & bound. Després de la seva implementació es farà una anàlisi de complexitat i comparativa entre algorismes.

La implementació dels algoritmes es realitza en una aplicació C ++, anomenada "GraphApplication", basada en diàlegs creats amb MFC (Microsoft Foundation Classes) i compilada amb Visual Studio 2019 amb el mòdul MFC. Per obtenir el màxim rendiment l'aplicació es compilarà per 64bits. Aquesta aplicació ja té una representació per als grafs i s'encarrega de les funcions de lectura, escriptura, visualització i creació aleatòria de grafs i visites.

# GraphApplication. Interfície d'usuari

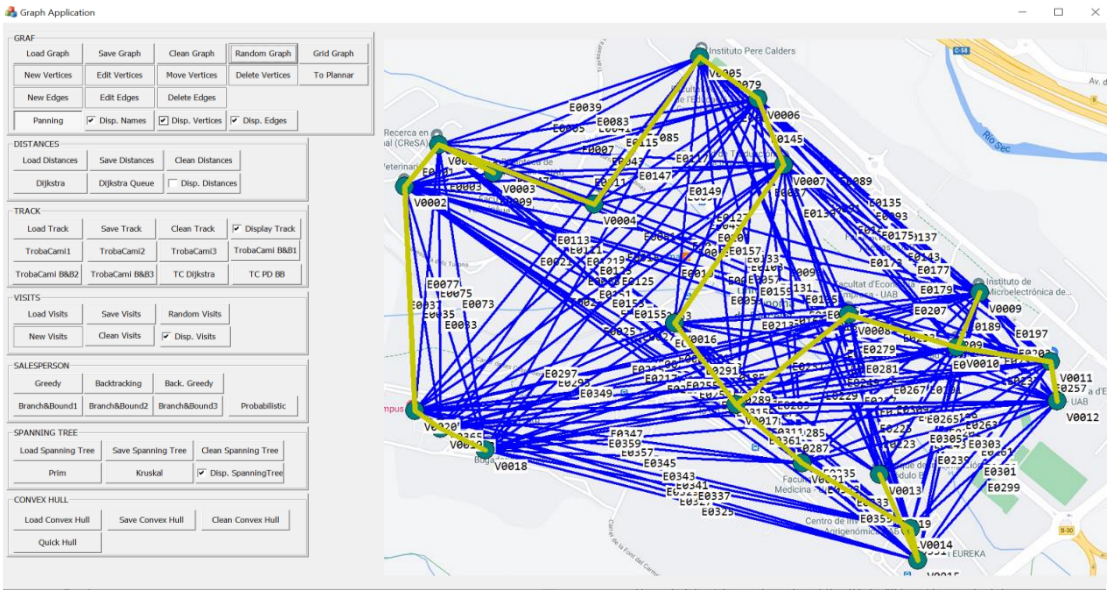


*Figura 1. Mostra de la interfície*

Aquesta interfície té diversos blocs segons la seva funcionalitat:

1. GRAF: El primer bloc conté tots els botons que permeten crear, eliminar, llegir i escriure grafs. També permet editar grafs, afegint i eliminant vèrtexs i arestes i visualitzar noms, vèrtexs i/o arestes.
2. DISTANCES: El segon bloc agrupa els botons relacionats amb les distàncies entre vèrtexs, i els algorismes per calcular-les.
3. TRACK: Aquest bloc agrupa les funcionalitats de la lectura i escriptura de camins
4. VISITS: aquest bloc agrupa les funcionalitats de les visites: creació, edició, lectura i escriptura de visites
5. SALESPERSON: Aquest bloc conté tots els botons dels algorismes que s'han de programar en aquest projecte
6. SPANNING TREE: Aquest bloc no s'utilitzarà en aquest projecte, però serveix per llegir, escriure i calcular arbres de cobertura mínima.
7. CONVEX HULL: Aquest bloc no s'utilitzarà en aquest projecte, però serveix per llegir, escriure i calcular convex hulls.

Per controlar la visualització del graf tenim les següents funcionalitats:

- Botó Panning activat: Moure graf arrossegant amb el botó esquerre del ratolí
- Rodeta polsada: Moure graf arrossegant amb la rodeta.
- Girar rodeta: zoom.
- Doble pulsació rodeta:  zoom a veure tot el graf dintre de la finestra.
- Ctrl-0: zoom a veure tot el graf dintre de la finestra
- Ctrl-Alt-0: zoom a mida real
- +: zoom fer més gran.

- -: zoom fer més petit

Quan executeu l'aplicació també s'obrirà una finestra que visualitza què está fent el programa i en la que podeu escriure mitjançant l'stream cout de C++.

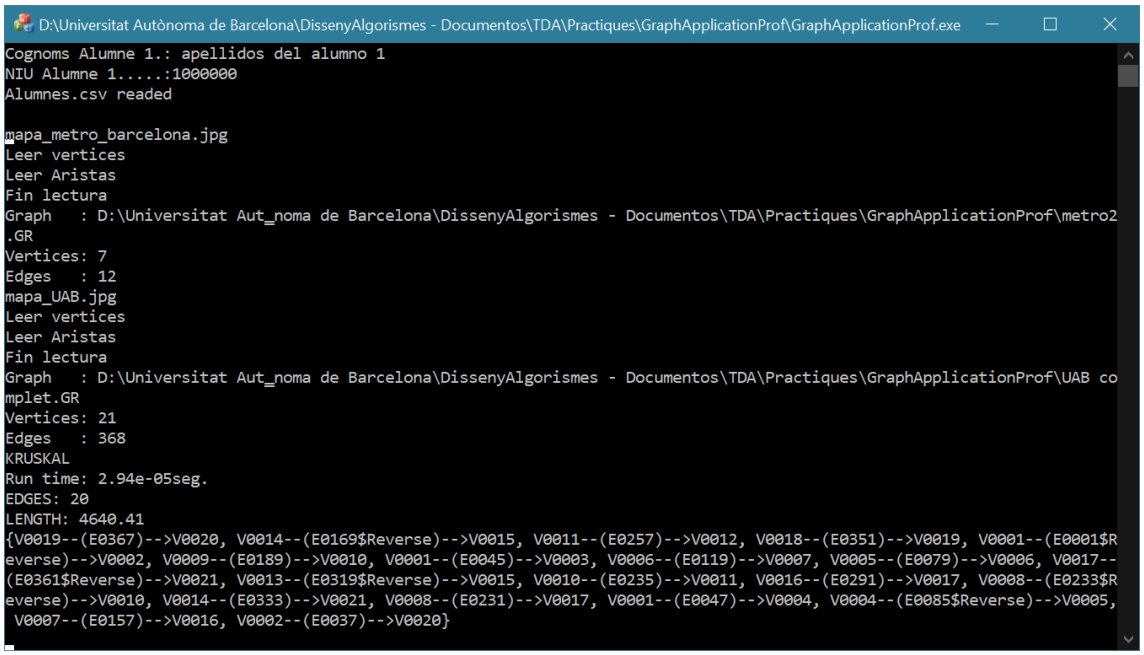


*Figura 2. Consola*

# Estructures de dades

## Representació del Graf

El graf amb el qual treballarem serà un graf els vèrtexs del qual són punts del pla real. El pes d'una aresta que connecta dos vèrtexs serà la distància euclidiana entre els dos vèrtexs.

D'aquesta manera, representarem el graf com una llista de vèrtexs i una llista d'arestes.

- Cada vèrtex té la llista d'apuntadors a les arestes que surten d'ell.
- Cada aresta apunta al vèrtex origen d'on surt i al vèrtex destí on arriba.
- En els grafs no dirigits cada aresta es representa com dues arestes dirigides en sentits contraris.
- Un graf pot tenir una imatge de fons associada per la visualització gràfica.
- Els grafs es guarden en fitxers amb extensió .GR

```
class CGraph {
public:
```

```
  list<CVertex> m_Vertices; // llista de vèrtexs
  list<CEdge> m_Edges; // llista d'arestes
  bool m_Directed; // graf dirigit o no dirigit
  string m_Filename; // Fitxer d'on s'ha llegit el graf
  string m_BackgroundFilename; // Fitxer de la imatge de fons
  CVImage* m_pBackground; // imatge de fons o NULL
};
```

## Vèrtex

- Cada vèrtex té un nom únic que no es pot repetir per altres vèrtexs.
- La longitud de l'aresta és la distància entre els vèrtexs que uneix. Està calculada en el camp `m_Length`.
- Les arestes de grafs no dirigits es representen amb dos `CEdge` en sentits contraris. El nom del `CEdge` en sentit contrari és `nom$Reverse`
- Els atributs propis necessaris per crear els grafs són els següents, tot i que la classe tingui més atributs (per als algorismes, per exemple):

```
class CVertex {
public:
  string m_Name; // Nom del vèrtex
  CGPoint m_Point; // Punt del pla del vèrtex.
  list<CEdge*> m_Edges; // llista d'arestes que surten del
vèrtex.
};
```

## Aresta

- Cada aresta té un nom únic que no es pot repetir per altres arestes.
- La longitud de l'aresta és la distància entre els vèrtexs que uneix. Està calculada en el camp `m_Length`.
- Les arestes de grafs no dirigits es representen amb dos `CEdge` en sentits contraris. El nom del `CEdge` en sentit contrari es `nom$Reverse`

```
class CEdge {
public:
  string m_Name; // Nom del edge
  double m_Length; // Longitud de l'aresta (distància entre
vèrtexs)
  CVertex* m_pOrigin; // Vèrtex d'on surt l'aresta
```

```
  CVertex* m_pDestination; // Vèrtex on arriba l'aresta
  CEdge* m_pReverseEdge; // aresta en sentit contrari per grafs
no dirigits
};
```

### Llista de visites

Una llista de visites és una llista de vèrtexs d'un graf. Cal tenir en compte que està associada a un graf que la suporta.

- Un camí que recorri les visites comença per la primera visita de la llista i acaba a l'última de la llista. Les visites intermèdies les pot visitar en qualsevol ordre.
- Un vèrtex només pot estar una única vegada a la llista de visites, excepte quan la llista és cíclica i comença i acaba al mateix vèrtex.
- Extensió dels fitxers de visites: .VIS

```
class CVisits {
public:
  list<CVertex*> m_Vertices; // Llista d'apuntadors als vèrtexs
del graf
  CGraph* m_pGraph; // graf que conté els vèrtexs.
};
```

### Camí

Un camí és una llista d'arestes d'un graf i està associat a un graf que el suporta.

- Les arestes d'un camí compleixen que al vèrtex on acaba una aresta, comença la següent.
- Extensió dels fitxers d'un camí: .TRK

```
class CTrack {
public:
  list<CEdge*> m_Edges;
  CGraph* m_pGraph;
};
```

## Material pel Projecte

A Caronte trobareu el fitxer del projecte "Projecte VS2019 SalesMan Vx.zip" (on x serà sempre l'última versió), que conté un projecte de Visual Studio 2019 amb totes les classes que necessiteu per fer el projecte. A l'arrel trobareu l'executable del vostre projecte (GraphApplication.exe) i que resulta de compilar-lo amb Visual Studio 2019.

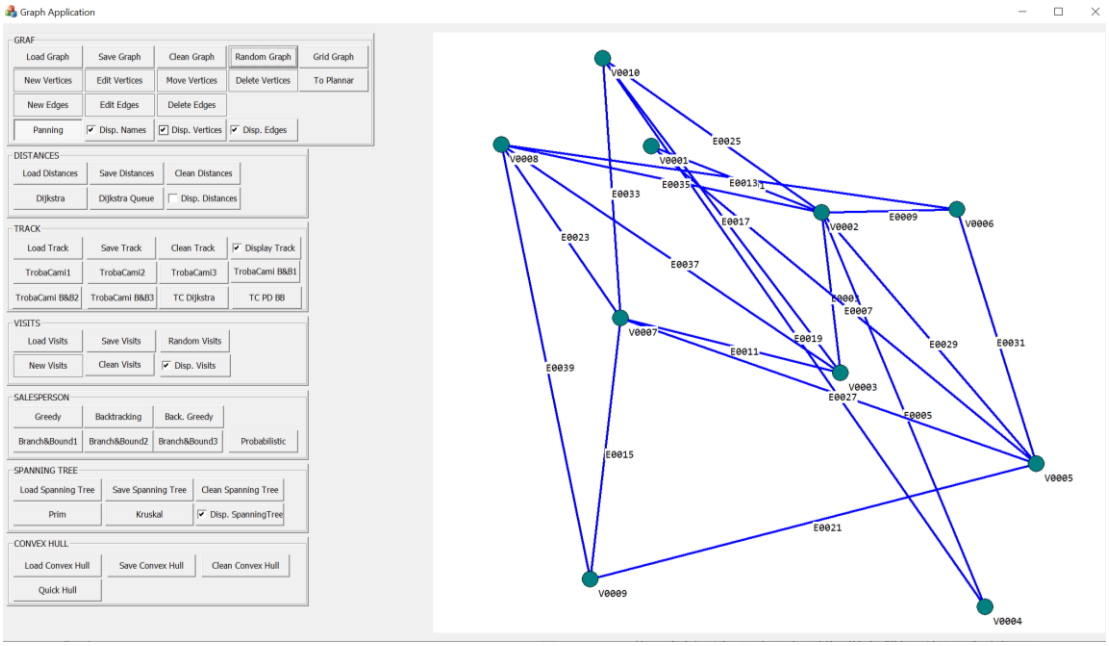


*Figura 3. Graph Application*

Des de la línia de comandes també es pot cridar al programa amb paràmetres perquè executi els algoritmes del projecte. Amb aquesta opció (que s'utilitzarà per a la correcció del projecte) es veu la identificació del grup, el fitxer del graf, les distàncies que s'han calculat i el temps que han trigat en segons. Com que aquesta sortida es farà servir per fer la correcció de les entregues és convenient que **la versió lliurada no imprimeixi res més**.

El projecte de la pràctica es pot compilar amb Visual Studio 2019 versió Community. En el cas que vulgueu fer una instal·lació al vostre ordinador del Visual Studio 2019 les opcions d'instal·lació mínimes són:

- A l'apartat de l'instal·lador "Càrrega de Treball": "Desenvolupament per a escriptori amb C ++"
- Activar el paquet " C++ MFC latest v142".

## Autocorrecció

Els lliuraments es faran mitjançant Caronte. Perquè us pugueu assegurar que el projecte funciona correctament abans del lliurament podeu provar-lo amb els correctors que adjuntem al fitxer .zip del projecte.

Ho podeu fer de dues maneres: amb l'executable (x64/Release/GraphApplication.exe) per poder jugar amb diferents grafs i/o amb els que criden el programa amb paràmetres (*.bat) per comparar les vostres sortides amb les nostres. Tingueu en compte que podeu executar en mode debug i en mode release. El mode debug és més laxe i també menys eficient que el mode release, així que us recomanem que ho feu en mode release. En aquest cas, tingueu en compte que totes les variables han d'estar inicialitzades correctament.

També podeu executar el fitxer GraphApplicationProf.exe sobretot per cal·librar els temps de referència.

## Proves

Un cop realitzades les implementacions es pot provar el projecte amb els fitxers que us proporcionem i verificar que el resultat és correcte amb els camins que també us proporcionem. El vostre resultat no ha de ser exactament el camí indicat en aquests fitxers, però ha de complir les següents condicions:

- Ha de ser d'igual o menor longitud que el camí corresponent al resultat
- El camí ha de passar per tots els vèrtexs de VisitsN.txt i complir que el camí comença al primer vèrtex de VisitsN.txt i acaba a l'últim vèrtex de VisitsN.txt.
- El camí serà una seqüència de vèrtexs $v_1, v_2, \dots, v_n$ que compleix que tot $v_i$pertany al graf i que $v_i$ i $v_{i+1}$ estan connectats per una aresta.

# Entregues

Al fitxer que conté el projecte hi haureu d'afegir el cos de les funcions corresponents als lliuraments que us anem demanant al llarg del curs. L'avaluació del projecte la realitzarem amb els fitxers de grafs que us hem proporcionat per comprovar que el programa funciona correctament. La sortida del programa s'utilitzarà per fer la correcció, així que **el programa no pot imprimir res més per la sortida estàndard**.

Cada projecte que es lliuri estarà identificat amb els noms i NIAs dels dos membres del grup. Al fitxer `GraphApplication.cpp` cal omplir les següents variables per identificar els membres del grup que presenten el projecte:

```
CString NombreAlumno1 = "nombre del alumno 1";
CString ApellidosAlumno1 = "apellidos del alumno 1";
CString NIAAlumno1 = "0000000"; // NIA alumno1

// No rellenar en caso de grupo de un alumno
CString NombreAlumno2 = "nombre del alumno 2";
CString ApellidosAlumno2 = "apellidos del alumno 2";
CString NIAAlumno2 = ""; // NIA alumno2
```

**Aquesta informació s'utilitzarà per posar les notes.**

Els lliuraments es realitzaran a Caronte. El fitxer per lliurar serà el fitxer "Projecte VS2019 SalesMan Vx.zip" amb el codi afegit. **L'estructura de directoris ha de ser la mateixa** que hi ha al fitxer "Projecte VS2019 SalesMan Vx.zip" de Caronte que heu utilitzat per fer el projecte. En cas que s'hagi de lliurar material addicional es posarà en el directori arrel de la pràctica. Abans de comprimir el projecte per fer el lliurament cal que executeu el fitxer "CleanProject.bat" perquè el projecte no pesi massa (feu-ho amb el projecte tancat per tal que pugui esborrar fitxers).

# Avaluació

Per fer la correcció del projecte farem proves amb els fitxers del directori TestsSalesman. Una part de la nota serà proporcional al nombre de test superats correctament i una altra part

dependrà del temps que tarda la pràctica en executar els tests. A més, hi haurà un punt extra en el cas que els temps siguin millors que els nostres. Si en algun cas cal lliurar un informe, aquest també comptarà per la nota del lliurament.

La fórmula general per calcular la nota de cada entrega serà la següent:

*NotaEntrega=0.8*NotaTests+0.2*NotaTemps+1*(si els temps milloren els nostres)*

On:

- $NotaTests = \frac{\#TestsOK}{\#TestsTotals}$
- $NotaTemps = 10 * (1 - \frac{t_a - t_r}{4t_r - t_r})$
- $t_a$ és la suma dels temps de l'alumne de totes les proves d'una entrega
- $t_r$ és la suma dels temps de referència de totes les proves d'una entrega

Consideracions:

1. En el cas que una prova *i* no es superi, $t_a^i = 4t_r^i$
2. En el cas que en una prova $t_a^i > 4t_r^i$ , el seu temps es considerarà $4t_r^i$
3. Aquelles entregues en les que $t_a \leq t_r$ tindran un punt extra directe.
4. Els temps es mesuraran en el PC del professor. Com poden canviar hi ha una calibració del corrector a partir d'executar la solució del professor.
5. L'entrega d'anàlisi serà un informe on es comparin els resultats de les implementacions de les entregues de backtracking, greedy i branch&bound.

## Recuperació

Els lliuraments de les pràctiques són acumulatius. Això permet recuperar o pujar la nota d'un lliurament en qualsevol lliurament posterior. En el cas d'una recuperació, la nota es calcularà de la següent forma:

notaActualitzada=(max(notaRecuperacio,notaEntrega)-notaEntrega)*0.8+notaEntrega

On notaActualitzada és la nota que tindrà l'entrega recuperada, notaEntrega és la nota que es va obtenir a l'entrega abans de la seva data límit i notaRecuperacio és la nota del lliurament posterior a la seva data límit. Si es fan vàries entregues de recuperació, s'escull la màxima nota d'aquestes.

# Lliurament 1. Greedy i algorisme de Dijkstra

L'objectiu d'aquest lliurament consisteix en resoldre el problema del viatjant de comerç utilitzant un algoritme greedy. La solució que buscarem serà aproximada i es basarà en l'heurística que fa que quan estiguem en una ciutat, anem a la ciutat més propera de les que hàgim de visitar. Aquesta heurística permet utilitzar el següent algoritme (greedy):

1. Començar per la ciutat d'origen
2. Mentre hi hagi ciutats per visitar excepte la ciutat destí repetir:
    a. Cercar quina és la ciutat de les que hem de visitar més propera a la que estem (en aquesta cerca no incloem la ciutat destí ja que sempre ha de ser l'última a la que es vagi)
    b. Anar a la ciutat més propera
    c. Treure la ciutat on estem de les que hem de visitar
3. Anar a la ciutat destí.

La ciutat origen és la primera de la llista de `CVisits` i la de destí és l'última. Les altres ciutats de la llista es podran visitar en qualsevol ordre.

Al fitxer "Greedy.cpp" trobareu la funció:

```
CTrack SalesmanTrackGreedy(CGraph& graph, CVisits &visits)
```

en la que haureu d'implementar l'algoritme explicat abans.

Per trobar la ciutat més propera primer haureu d'implementar **l'algorisme de Dijkstra**. Com que aquest algoritme només dóna distàncies i no camins caldrà tenir-ho en compte per tenir camins. Considereu l'ús del camp `m_pDijkstraPrevious` per crear el camí més curt. Aquest apuntador apuntarà en sentit contrari el recorregut del camí creat des del vèrtex `pStart`.

Tenint un graf ponderat d'N nodes, i sent x el node inicial, un vector D de mida N guardarà al final de l'algoritme les distàncies des d'x a la resta dels nodes.

Com podeu visualitzar en aquest vídeo, els passos són (extret de https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra):

1. Inicialitzar totes les distàncies a D amb un valor infinit relatiu ja que són desconegudes al principi, exceptuant la d'x que s'ha de col·locar a 0 a causa de que la distància de x a x seria 0.

2. Sigui a = x (prenem a com a node actual).

3. Recorrem tots els nodes adjacents d'a, excepte els nodes marcats, anomenarem a aquests nodes no marcats $v_i$.

4. Per al node actual, calculem la distància temptativa des d'aquest node als seus veïns amb la següent fórmula: $dt(v_i) = d_a + d(a, v_i)$. És a dir, la distància temptativa del node $v_i$ és la distància que actualment té el node al vector D + la distància des de l'esmentat node 'a' (l'actual) al node '$v_i$'. Si la distància temptativa és menor que la distància emmagatzemada al vector, actualitzem el vector amb aquesta distància temptativa. És a dir: Si $dt(v_i) < D_{v_i} \rightarrow D_{v_i} = dt(v_i)$

5. Marquem com complet el node a.

6. Prenem com a pròxim node actual el de menor valor en D que no s'hagi visitat i tornem al pas 3 mentre hi hagi nodes no marcats.

Un cop acabat l'algoritme, D estarà completament ple amb les distàncies d'x a qualsevol altre node del graf.

Al fitxer "Dijkstra.cpp" trobareu les funcions:

```
void Dijkstra(CGraph& graph, CVertex *pStart)
void DijkstraQueue(CGraph& graph, CVertex *pStart)
```

A la primera funció heu d'implementar l'algorisme de Dijkstra amb cerca seqüencial del vèrtex amb mínima distància (pas 6 de l'algorisme) i a la segona funció la versió optimitzada de l'algoritme que utilitza una cua amb prioritat per cercar el vèrtex amb mínima distància.

## Consells per fer la implementació dels algoritmes de Dijkstra

• Cal implementar una funció que generi les dades dels temps automàticament. Així es podran repetir els experiments de forma més fàcil, ràpida i controlada.

• Per implementar l'algoritme de Dijkstra amb cua amb prioritat podeu utilitzar la classe priority_queue de la STL.

• Per calcular els temps d'execució cal utilitzar un rellotge d'alta velocitat. La funció Clock () de la pràctica us dóna aquest servei amb precisió de nanosegons.

• Els temps d'execució no s'ajustaran exactament a les complexitats teòriques.

• Els grafs han de ser grans perquè els temps d'execució siguin realment representatius.

• Cal considerar la granularitat del rellotge de l'ordinador.

• L'ordinador no pot fer res més quan es calculen temps per evitar que qualsevol altre procés ens afecti els mesuraments.

### Lliurament

Abans de fer el lliurament recordeu executar el fitxer `CleanProject.bat` dins del directori de la solució. Aquesta comanda eliminarà els fitxers temporals creats per VS2019 i reduirà la mida de la solució perquè es pugui pujar a Caronte.

El lliurament constarà del projecte del viatjant amb les funcions:

```
void CGraph::Dijkstra(CVertex *pStart)
void CGraph::DijkstraQueue(CVertex *pStart)
CTrack CGraph::SalesmanTrackGreedy(CVisits &visits)
```

implementades i es farà via Caronte.

## Lliurament 2. Backtracking

L'objectiu d'aquest lliurament consisteix en resoldre el problema del viatjant de comerç utilitzant algoritmes basats en backtracking.

S'implementaran dos algorismes basats en backtracking. El primer serà un algoritme de backtracking pur i el segon combinarà backtracking amb greedy. Aquests algoritmes hauran de donar el camí més curt possible, és a dir, la solució òptima.

En aquest enunciat només es dóna una guia sobre com han de ser aquests algoritmes, però els detalls concrets els haurà de deduir l'estudiant.

### Algorisme de Backtracking Pur

Aquest algorisme el podem resoldre des de dos punts de vista, per vèrtexs o per arestes.

#### Solució per vèrtexs

La cerca del camí òptim d'un vèrtex $V_O$ a un vèrtex $V_d$ consisteix en

1. Des del vèrtex $V_O$ seguim per una aresta a un veí $V_1$
2. Des de $V_1$ anem a un veí, $V_2$, de $V_1$
3. i així successivament.

El pas endavant de l'algorisme consisteix en anar a un vèrtex veí des del vèrtex actual. En cas de no poder anar a cap veí tornem enrere i seleccionem un altre veí. Tindrem un camí de $V_O$ a $V_d$ quan el vèrtex seleccionat coincideix amb el vèrtex destí $V_d$.

**Observació:** per evitar bucles infinits, quan seleccionem un vèrtex, no podem passar per un vèrtex que ja hem passat dins d'aquell camí parcial. Això no vol dir que no es puguin repetir en un altre camí parcial.

Una possible representació per saber per quins vèrtexs hem passat és una pila:

- Posem un vèrtex a la pila quan arribem a ell en un pas endavant
- Traiem un vèrtex de la pila quan fem un pas enrere.

Falta controlar què passa quan arribem a un vèrtex que hem de visitar: Si el vèrtex a visitar no l'havíem visitat fins ara, haurem de considerar que, al posar-lo a la pila, podem tornar a passar per tots els vèrtexs anteriors.

D'aquesta forma tenim:

- Pas endavant:
    - Per cada veí del vèrtex actual veure si es pot passar per ell:
        - Verificar que no està a la pila abans de l'últim vèrtex a visitar pel que es passa per primera vegada (vèrtex que s'havia de visitar).
        - Si es pot passar:
            - Anar al vèrtex i posar-lo a la pila. Marcar-lo com a vèrtex que s'havia de visitar si està a la llista de vèrtexs per visitar i no s'ha passat encara per ell.
            - Crida recursiva amb el vèrtex actual (al que acabem d'anar).
        - Si no es pot passar, anar al següent veí
    - Si no hi ha més veïns, fer un pas enrere.
- Pas enrere:
    - Treure un vèrtex de la pila i seleccionar com actual l'últim de la pila.

### *Optimitzacions de la representació del graf*

- Evitar cercar a la llista de vèrtexs a visitar. Marcar els vèrtexs a visitar amb un booleà al graph. Així caldrà buscar el vèrtex a la llista de visites.
- Guardar la informació del recorregut fet en els vèrtexs del graf
    - Per guardar el recorregut fet utilitzem una pila com en el cas anterior.
    - Les dades que conté aquesta pila per cada vèrtex les incorporem en els propis vèrtexs del graf. Així evitem haver de buscar els vèrtexs a la pila per obtenir aquesta informació.
    - Per saber si un vèrtex podem passar per ell o no, considerarem en concepte de tram del camí
        - Un tram és la part del camí que va d'un vèrtex a visitar a un altre vèrtex a visitar. Així si hem de visitar n vèrtexs, el camí estarà format per n-1 trams.
        - Podem guardar a cada vèrtex el número del tram al que pertany. Compte, com podem passar més d'una vegada per cada vèrtex, aquest pot pertànyer a més d'un tram.

### Solució per vèrtexs

Aquesta forma de resoldre el problema del viatjant de comerç simplifica molt la codificació, ja que considera que el camí que s'està construint no pot passar més d'una vegada per la mateixa aresta en el mateix sentit. Com a la representació del graf que estem utilitzant les arestes tenen sentit, no podrem passar més d'una vegada per la mateixa aresta.

*Consideracions:*

Per saber si s'ha passat per una aresta, podem afegir un camp a CEdge que ho indiqui.

Solució 1: Podem cercar un camí que va del vèrtex origen al vèrtex destí de les visites sense considerar que passi per les visites intermitges. Així, només considerarem que el camí és una solució quan estem al vèrtex destí, i després de verificar que passa per a totes les visites.

Solució 2: Podem cercar un camí que va del vèrtex origen al vèrtex destí i en la seva construcció ja comptem els passos per visites intermitges. En aquest cas, només considerarem que el camí és una solució quan el comptador de visites intermitges és igual al total de visites i estem al vèrtex destí.

Farem un pas de backtracking (no continuar pel mateix camí) quan:

- Ja hem explorat totes les arestes que surten del vèrtex on acaba el camí.
- Quan la longitud del camí actual és més gran que la del millor camí solució que hem trobat abans.
- Quan trobem una solució.

## Algorisme Combinat Backtracking-Greedy

En aquest algoritme aprofitarem l'algoritme de Dijkstra per buscar el camí més curt entre dos vèrtexs a visitar. Així l'espai de cerca que explorarem amb backtracking es reduirà a l'ordre en què hem visitat els vèrtexs de la llista de visites. D'aquesta manera, si hem de visitar n vèrtexs, el primer node de l'arbre de cerca tindrà n fills que corresponen als n vèrtexs a visitar. En el següent nivell, els nodes de l'arbre tindran n-1 fills ja que ja s'ha visitat un vèrtex, i així, successivament fins a completar els n nivells de l'arbre. Com explorar tot l'arbre pot ser molt costós, es pot realitzar un forward-checking  basat en que, si el millor camí que hem trobat és més curt que la longitud d'el camí que estem creant, no cal completar el camí que estem creant.

### Guies d'implementació

- Guardar en un array de dos dimensions tots els camins òptims entre vèrtexs a visitar.
- El camí de $vv_i$ a $vv_j$ es guardarà a la posició (i,j) de l'array.
- Cridar a Dijkstra posant com a origen cadascun dels vèrtexs a visitar. Guardar els camins generats a l'array.
- Convertir els vèrtexs a visitar en índexs d'aquest array.
- Implementar el backtracking de forma que treballi sobre un array d'índexs de vèrtexs (tupla).
- Tallar la cerca quan el camí que estem generant és més llarg que el camí més curt que hem trobat fins el moment.

## Lliurament

Abans de fer el lliurament recordeu executar el fitxer `CleanProject.bat` dins del directori de la solució. Aquesta comanda eliminarà els fitxers temporals creats per VS2019 i reduirà la mida de la solució perquè es pugui pujar a Caronte.

El lliurament constarà del projecte del viatjant amb les funcions:

```
// SalesmanTrackBacktracking ================================================

CTrack CGraph::SalesmanTrackBacktracking(CVisits &visits)
{
	CTrack track(this);
	return track;
}

// SalesmanTrackBacktrackingGreedy ==========================================

CTrack CGraph::SalesmanTrackBacktrackingGreedy(CVisits &visits)
{
	CTrack track(this);
	return track;
}
```

implementades i es farà via Caronte.

Degut a la complexitat de l'algorisme de backtracking pur, els grafs a testejar no poden ser excessivament grans.

# Lliurament 3. Branch & bound

L'objectiu d'aquest lliurament consisteix en resoldre el problema del viatjant de comerç utilitzant algoritmes basats en branch & bound.

S'implementaran tres algorismes basats en branch & bound. La diferència serà l'heurística utilitzada per dirigir el recorregut per l'arbre de l'espai de solucions. Aquests tres algoritmes hauran de trobar el camí més curt possible, és a dir, la solució òptima. Les heurístiques utilitzades seran:

1. Completar sempre el camí parcial més curt dels que s'hagin creat. En el cas d'haver trobat un camí complet, es podaran els camins parcials més llargs que el millor camí complet que s'hagi trobat fins al moment.
2. Per a cada camí parcial calcular una cota inferior de la longitud que tindria el camí complet. Completar el camí parcial pel qual la cota inferior per a la longitud total sigui la menor dels camins parcials. En el cas d'haver trobat un camí complet, es podaran els camins parcials la cota inferior dels quals sigui major que la longitud del millor camí complet que s'hagi trobat. La cota inferior pel camí complet serà la suma de la longitud del camí parcial més el camí més curt de cadascun dels vèrtexs que falten per visitar al seu veí més proper. Aquest veí serà uns dels vèrtexs a visitar. Per reduir el temps de

càlcul, el veí més proper serà qualsevol vèrtex dels que es troben la llista `visits` tot i que ja hagi visitat.

3. Heurística proposada per l'alumne. Aquesta heurística s'explicarà a la sessió presencial dedicada als algorismes de B&B pel projecte.

Per a la implementació de l'algoritme de branch & bound es pot utilitzar una cua amb prioritat on es guardaran les solucions parcials que es poden completar. És important recordar que aquest algoritme ha de trobar la solució òptima.

## Lliurament

Abans de fer el lliurament recordeu executar el fitxer `CleanProject.bat` dins del directori de la solució. Aquesta comanda eliminarà els fitxers temporals creats per VS2019 i reduirà la mida de la solució perquè es pugui pujar a Caronte.

El lliurament constarà del projecte del viatjant amb les funcions:

```
// SalesmanTrackBranchAndBound1 ================================================

CTrack CGraph::SalesmanTrackBranchAndBound1(CVisits &visits)
{
	return CTrack(this);
}

// SalesmanTrackBranchAndBound2 ================================================

CTrack CGraph::SalesmanTrackBranchAndBound2(CVisits &visits)
{
	return CTrack(this);
}

// SalesmanTrackBranchAndBound3 ================================================

CTrack CGraph::SalesmanTrackBranchAndBound3(CVisits &visits)
{
	return CTrack(this);
}
```

implementades i es farà via Caronte.

# Lliurament 4. Anàlisi de complexitat algorísmica

L'objectiu d'aquest lliurament consisteix en analitzar els algorismes implementats durant tot el projecte. Només es lliurarà un informe amb els resultats de l'anàlisi, no cal lliurar cap programa. Els algorismes a analitzar són:

- Algorisme greedy
- Algorisme de backtracking pur
- Algorisme de backtracking + greedy
- Algorisme de branch & bound amb l'heurística 1
- Algorisme de branch & bound amb l'heurística 2
- Algorisme de branch & bound amb l'heurística 3

L'informe s'organitzarà per respostes a la pregunta exposada a l'enunciat. Cada resposta es justificarà amb una o més gràfiques que la demostrin experimentalment. Les dades d'aquestes gràfiques sortiran d'aplicar els algorismes sobre problemes generats aleatòriament i els paràmetres seran nombre de vèrtexs del graf, nombre d'arestes, nombre de vèrtexs a visitar.

Les preguntes a respondre són les següents:

1. Com varia el temps d'execució dels algorismes en variar el nombre de vèrtexs? Constant, logarítmic, lineal, exponencial, factorial, etc.
    - Resposta. Una gràfica amb les corbes dels 6 algorismes. En el cas que no es pugui veure bé alguna corba en la primera gràfica, caldrà fer una altra gràfica on només es vegin les corbes que no es veien bé.

2. Com varia el temps d'execució dels algorismes en variar el nombre d'arestes? Constant, logarítmic, lineal, exponencial, factorial, etc.
    - Resposta. Una gràfica amb les corbes dels 6 algorismes. En el cas que no es pugui veure bé alguna corba en la primera gràfica, per a una altra gràfica on només es vegin les corbes que no es veien bé.
3. Com varia el temps d'execució dels algorismes en variar el nombre de visites? Constant, logarítmic, lineal, exponencial, factorial, etc.
    - Resposta. Una gràfica amb les corbes dels 6 algorismes. En el cas que no es pugui veure bé alguna corba en la primera gràfica, per a una altra gràfica on només es vegin les corbes que no es veien bé.

4. Com empitjora el resultat de l'algorisme greedy respecte la solució òptima en variar nombre de vèrtexs, arestes i visites? ¿Per què no varia en modificar el nombre d'arestes i visites?
    - Resposta. Una gràfica on s'han fixat el nombre de vèrtexs i arestes. Només es varia el nombre de visites.
    - L'empitjorament d'un resultat es pot mesurar com el tant per cert de longitud de més que té una solució respecte l'òptima.

5. Quant ha de podar un algorisme de Branch & bound (1,2,3) per ser millor que un algorisme de backtracking greedy? Considerar el nombre de nodes de l'arbre de cerca que recorre cada algorisme i quant temps dedica a cada node. Veure gràficament quan s'aconsegueix una millora de temps considerant els nodes de l'arbre de cerca visitats.
    - Resposta. Per respondre a aquesta pregunta fixar el nombre de vèrtexs i arestes del graf. Només variar el nombre de visites.
    - Considerar gràfiques amb temps i gràfiques amb nodes de l'arbre de solució visitats. Els nodes de l'arbre de cerca seran les solucions parcials generades pels algorismes de Branch & bound i les crides recursives de l'algorisme de Backtracking.

*Observacions:*

- Es pot justificar la fórmula de la complexitat de les implementacions per parts, és a dir, variant només el nombre de vèrtexs o variant només el nombre d'arestes del graf.
- Una gràfica demostra que la complexitat de la implementació es correspon amb la complexitat teòrica si la forma de la corba és semblant. Per tant, cal posar com a mínim 10 punts per fer la gràfica i l'escala de representació ha de permetre veure bé la forma de la corba.

- Els valors dels eixos de les gràfiques han de seguir una progressió lineal, per exemple, 10, 20, 30, 40 ... No és vàlid posar valors com 10, 15, 50, 100 ja que deformen la corba representada.
- Per fer les gràfiques és convenient crear nous grafs amb la funció "`Random Graph`" amb el nombre de vèrtexs i arestes convenient per fer les gràfiques.
- El codi serà més ràpid si es compila en versió release de 64 bits.