Un **algoritmo euristico** (o **una euristica**) è un algoritmo progettato per risolvere un problema più velocemente, nel caso in cui i metodi classici siano troppo lenti nel calcolo (ad esempio, in caso di elevata complessità computazionale) o per trovare una soluzione approssimata, nel caso in cui i metodi classici falliscano nel trovare una soluzione esatta. Il risultato viene ottenuto cercando di equilibrare gli obiettivi di maggiori ottimizzazione, completezza, accuratezza e velocità di esecuzione. https://it.wikipedia.org/wiki/Algoritmo_euristico

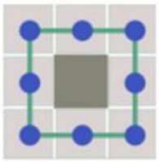
Una euristica è una soluzione vicina all'algoritmo corretto che magari impiega troppo tempo o troppo spazio.

Per i problemi della vita reale non è nota la soluzione efficiente ma di solito troviamo una euristica (organizzare le tappe di un viaggio, le ricerche in biologia, risolvere un Sudoku, …).

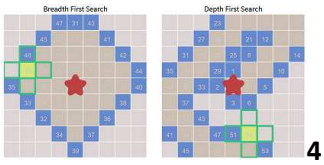
https://www.redblobgames.com/pathfinding/a-star/introduction.html

Problema del labirinto : trovare il cammino minimo da ★ a ✕.

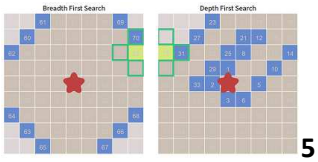
p.18-19 Un labirinto può essere rappresentato con un grafo:



ad ogni cella corrisponde un vertice, le celle adiacenti sono collegate da un arco; da ogni cella sono possibili solo 4 spostamenti: verso l'alto, il basso, a sinistra e a destra; inutile rappresentare i muri nel grafo perchè sono vertici senza collegamenti.

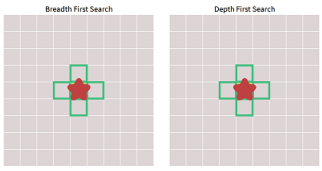


4

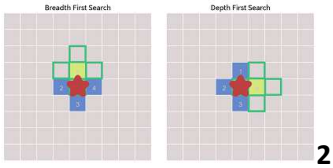


5

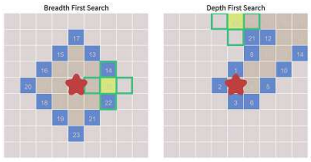
Per risolvere un labirinto si possono usare gli algoritmi BFS e DFS che su una esplorazione esaustiva di un grafo hanno tempi equivalenti: **Comparing BFS and DFS**



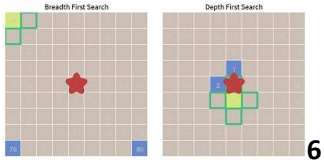
1



2

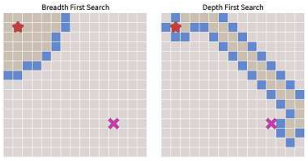


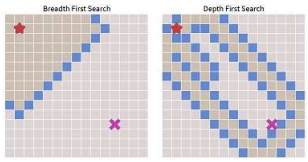
3

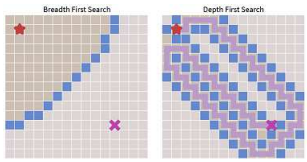


6

Per un labirinto invece è **più efficiente l'algoritmo BFS** https://cs.stanford.edu/people/abisee/tutorial/bfsdfs.html



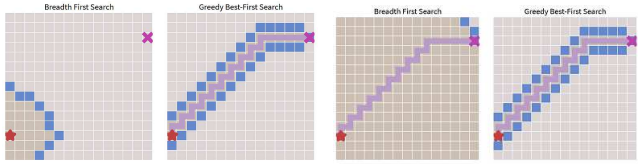




BFS si allarga espandensosi in larghezza ma termina dopo il DFS che si allunga in profondità ma con percorso più lungo.

Greedy Best First Search

https://cs.stanford.edu/people/abisee/tutorial/greedy.html

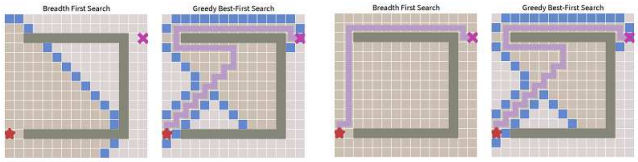


p.22-23 Gli algoritmi dei cammini minimi possono essere meno efficienti di una soluzione euristica. L'algoritmo **BFS può essere migliorato usando un approccio Greedy**.

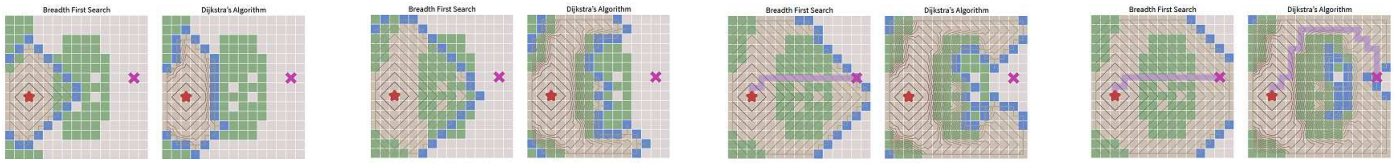
Invece di considerare ed esplorare tutti i vertici adiacenti ad un vertice si può considerare solo quello che ci avvicina all'obiettivo, cioè tra i 4 spostamenti possibili, si può scegliere il vertice a distanza minima dalla destinazione.

Ma non fornisce la soluzione migliore in questo caso →

Con il muro a C capovolta l'euristica Gredy BFS fornisce una soluzione più veloce ma il percorso è più lungo.



Confrontiamo ora l'**algoritmo BFS** e quello **di Dijkstra** applicato ad una mappa geografica dove le zone con rilievi o foreste sono rappresentate da archi di peso maggiore di quelli su zone pianeggianti o erbose (fitti boschi o dislivelli da superare rallentano un viaggiatore); l'alg. Dijkstra viene utilizzato per problemi reali, trova il percorso meno costoso (in questo caso che richiede minor sforzo fisico/tempo) percorso che non è necessariamente quello più corto.

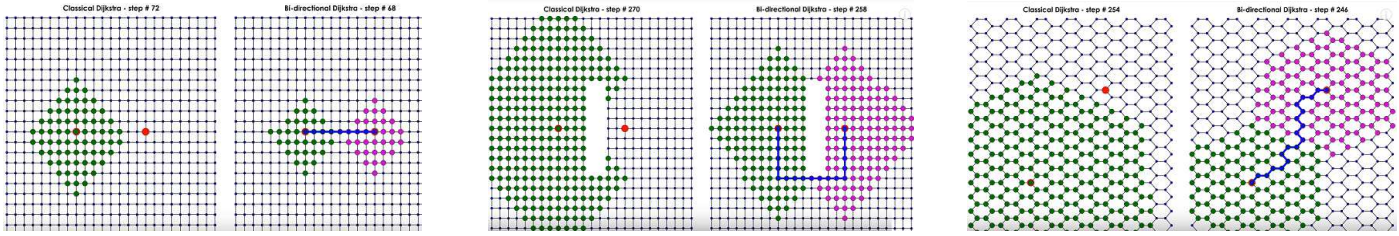


E il tempo di esecuzione è maggiore di quello del BFS. https://cs.stanford.edu/people/abisee/tutorial/dijkstra.html

Per un labirinto non interessano i percorsi minimi dalla sorgente a tutti gli altri nodi, ma solo verso la destinazione.

Per migliorare i tempi si ricorre al **Dijkstra bidirezionale**: si iniziano contemporaneamente le ricerche del cammino minimo dalla sorgente e dalla destinazione fermandsi quando si incontrano (***forward search*** –si considerano gli archi uscenti dalla sorgente- e ***backward search*** – archi entranti nella destinazione). Confronto algoritmi mediante animazioni:

https://www.youtube.com/watch?v=8Jjdp6f7oaE griglia rettangolare griglia esagonale



il Dijkstra classico si ferma allo step #130 il classico con un muro si ferma allo step #406 il classico si ferma allo step #304

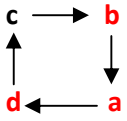
p.30 ATTENZIONE: il Dijkstra bidirezionale rischia di fornire una soluzione non corretta: il costo del percorso sxt è 10 mentre la soluzione corretta, sabt, costa solo 9.

esercizio: **problema Multi-Layer Dictionary (dictionary)**

il problema può essere modellato con un grafo:

- ogni vertice è una parola,
- un arco tra le parole a e b punta da **a** a **b** se nella definizione di b uso a: a → b

il **caso di test 3** è quello più chiaro:



N=4 parole nel dizionario, D=4 parole definite di seguito nel file

a è definita da **b** b → a **b** è definita da **c** c → b

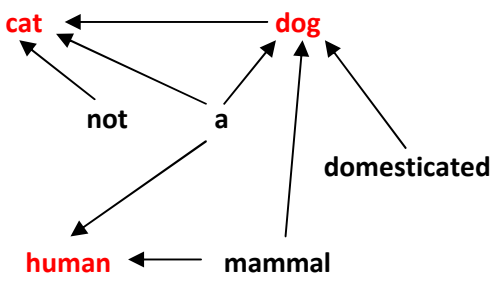
c è definita da **d** d → c **d** è definita da **a** a → d

è un grafo con un ciclo, quindi basta imparare una sola parola, ad esempio: **c**

---

le parole che non hanno archi entranti sono già definite nel dizionario;

nel **caso di test 1** le parole: **a not domesticated mammal** sono le 4 parole (minimo) necessarie a definire tutte le altre.



Si può partire dai nodi indipendenti e procedere per livelli; usando la tecnica Greedy: inseriamo nel set minimo i nodi con un alto grado di archi uscenti cioè parole usate in molte definizioni.

Gestione delle STRINGHE

Per alcuni problemi sulle stringhe la classe ***string*** di ***std*** library e gli array di char del C non hanno buone prestazioni.

Per l'**inserimento (*insert*)** o per la **cancellazione (*erase*)** di un carattere in una stringa lunga **n** devono scorrere di un posto i caratteri della parte superiore della stringa; nei casi peggiori può essere necessario far slittare tutti gli elementi – quindi complessità computazionale O(n). Analogamente per la **concatenazione** di stringhe (operatore **+** per ***string***) perchè occorre copiare tutti i caratteri di una o entrambe le stringhe nella stringa di destinazione O(n+m).

Per queste operazioni è preferibile l'uso di **liste di caratteri** (***linked lists***) oppure la tecnica dei ***tries*** che, però, per le operazioni di accesso/sostituzione di un carattere non sono indicate.

Per ogni problema va scelta la struttura dati più adatta: ***linked lists*** e ***tries*** sono più veloci per le operazioni indicate anche se utilizzano più spazio di memoria (per le OII lo spazio non è un problema, i tempi sono invece fondamentali)

linked lists

In C++ una lista concatenata si realizza con i puntatori https://it.wikipedia.org/wiki/Puntatore_(programmazione)

p.66 la struttura **cella** è composta da un campo **c** di tipo char e da un campo **next** di tipo puntatore a dati di tipo cella (**cella ***). Definita la struttura **cella** vengono definiti 2 puntatori a tipo cella, **first** e **last**, che puntano rispettivamente all'inizio (1^ carattere della stringa) e alla fine (ultimo carattere) della lista di caratteri da costruire.

I puntatori first e last inizialmente non puntano a nessun oggetto (puntatore a ***NULL*** si indica con ***nullptr*** dal C++11).

p.68 l'istruzione ***new cella*** crea a run-time un **nuovo elemento di tipo cella** che viene anche inizializzato con:

- 'a' per il campo **c** e con
- nullptr per il campo **next**

va contestualmente salvato l'indirizzo del nuovo elemento: **first = new cella** …. in un puntatore a tipo cella
Anche **last** viene modificato per puntare all'indirizzo dell'unico carattere presente nella stringa (last = first).

p. 71 viene creato un nuovo elemento di tipo cella: **new cella {'x',P->next};** anche in questo caso serve una variabile di tipo puntatore a tipo cella (***newcell***) per salvare l'indirizzo del nuovo elemento. Stavolta la definizione del puntatore *newcell* (**cella * newcell =** ….) viene fatta contestualmente alla creazione del nuovo elemento di tipo cella.

L'istruzione compatta **cella* newcell = new cella {'x',P->next};** equivale alle istruzioni:

cella* newcell; newcell = new cella; newcell->**c** = 'x'; newcell->**next** = P->next;

il simbolo -> specifica un campo di un dato strutturato indicato da un puntatore (*newcell* e *P* sono puntatori)
invece il simbolo **.** viene usato per specificare direttamente un campo di un dato strutturato

NOTA: indicando per brevità con ***elemento_h*** di tipo cella l'elemento che contiene nel campo c il valore 'h':
se si fosse definita la **variabile statica elemento_h** di tipo cella con l'istruzione cella elemento_h; si potevano assegnare i valori ai suoi campi con la sintassi di assegnazione di valore per le strutture: elemento_h.**c** = 'h'; elemento_h.**next** = nullptr;
I valori del nuovo elemento da inserire sono 'x' per il campo **c** e l'indirizzo attualmente contenuto nel campo **next** di *elemento_e* che punta ad *elemento_l* (il campo **next** di *elemento_e* contiene l'indirizzo di *elemento_l*).
Ma **P** punta a *elemento_e* quindi **next** di *elemento_e* può essere indicato con **P->next**

Negli esempi del prof. Prezza gli elementi di tipo cella vengono creati tutti dinamicamente, cioè a run-time.

ATTENZIONE: la memoria allocata dinamicamente nei progammi C/C++ andrebbe deallocata esplicitamente (delete)

esempio gestione **liste di caratteri**

il programma inserisce 4 caratteri (equivalenti alla stringa "ciao") nella lista con puntatori **F** (first) e **L** (last), definisce il puntatore **ptr** e scorre di alcune posizioni la lista; inserisce quindi il carattere 'd' e poi stampa l'intera lista (stringa).

strings.cpp

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  struct cella{
7
8      char c;
9      cella* next;
10
11 };
12
13 using lista = cella*;
14
15 void append(lista& F, lista& L, char c){
16
17     if(L == nullptr){
18
19         L = new cella {c,nullptr};
20         F = L;
21
22     }else{
23
24         L->next = new cella {c,nullptr};
25         L = L->next;
26
27     }
28
29 }
30
31 void print(lista F){
32
33     while(F){
34
35         cout << F->c;
36         F = F->next;
37
38     }
39
40 }
41
```

lista è un **alias** per puntatori a cella

vengono definiti ed inizializzati i puntatori **F** ed **L** di tipo *lista* ovvero di tipo puntatore a cella *(righe 63-64)*

si accodano alla lista 4 caratteri

la funzione *append* ha **2 parametri passati per riferimento** (i puntatori, che la funzione deve modificare) e **1 per valore** (il carattere da accodare);

se L è ancora uguale a *nullptr* (cioè se la lista è ancora vuota) sia L che F devono puntare al primo elemento che si sta creando (indirizzo di new cella);

altrimenti: il campo next dell'ultimo elemento ora in lista (L->next) deve puntare al nuovo elemento creato,
poi L può essere aggiornato con l'indirizzo dell'ultimo elemento creato

```
42 void insert(lista& F, lista& L, cella* ptr, char c){
43
44     if(not F){
45
46         append(F,L,c);
47
48     }else{
49
50         ptr->next = new cella{c,ptr->next};
51
52         if(ptr == L) L = ptr->next;
53
54     }
55
56 }
57
```

```
61 int main(){
62
63     lista F = nullptr;
64     lista L = nullptr;
65
66     append(F,L,'c');
67     append(F,L,'i');
68     append(F,L,'a');
69     append(F,L,'o');
70
71     cella* ptr = F;
72     ptr = ptr->next;
73     ptr = ptr->next;
74
75     insert(F,L,ptr,'d');
76
77     print(F);
78
79
80 }
```

nel *main*:

definito il puntatore **ptr** e impostato con l'indirizzo di inizio della lista contenuto in **F**, si scorre la lista di 2 posizioni: all'inizio **ptr** punta a *elemento_c*, alla fine punta ad *elemento_a*; quindi l'*elemento_d* viene inserito tra *elemento_a* ed *elemento_o*.

La funzione *print* scriverà: **ciado**

Quanto alla funzione *insert*: se la lista è vuota (F contiene il *null character,* zeri binari, e quindi **not F**, numero diverso da 0, risulta VERO) l'elemento viene accodato, ovvero inserito in prima posizione perchè la funzione *append* rileva anche **L** a NULL.

La gestione di una stringa con liste di caratteri è più complicata ma efficiente per l'inserimento e la cancellazione di un carattere (nel caso sia noto l'indirizzo) e per la concatenazione di stringhe (basta modificare 2 puntatori).

Anche le classi ***vector*** e ***string*** si espandono dinamicamente: (https://www.cplusplus.com/reference/vector/vector/) ma il costo dell'ampliamento è significativo; se si sono allocati ed utilizzati 1000 elementi, all'accodamento del successivo elemento lo spazio viene riallocato, raddoppiato di dimensioni (2000) e vengono spostati tutti gli elementi nella nuova zona di memoria (in realtà si procede per potenze di 2: prima allocazione 2 poi 4, 8, 16, …).

Il costo dell'ampliamento viene però **ammortizzato** sui singoli elementi e per la complessità di ogni operazione di ***push_back*** si considera un costo costante: O(1)

https://www.cplusplus.com/reference/vector/vector/push_back/

la complessità di ***insert*** e di ***erase*** invece è O(n) → https://www.cplusplus.com/reference/vector/vector/insert/

https://www.cplusplus.com/reference/vector/vector/push_back/ 110%

Complexity

Constant (amortized time, reallocation may happen).

If a reallocation happens, the reallocation is itself up to linear in the entire size.

Complexity **Linear on** the number of elements inserted (copy/move construction) plus **the number of elements after position (moving).**

per la gestione di string e vector vedere pag http://www.programmiamo.altervista.org/C/oop/oop11.html *e successiva*

esercizio: problema: join strings https://open.kattis.com/problems/joinstrings

dalla piattaforma di competitive programming **KATTIS** https://open.kattis.com/

N stringhe non vuote S1, S2…, N-1 operazioni: l'operazione ab concatena la stringa a_{esima} con la b_{esima} e la salva in a svuotando la stringa b l'alunno Lucio suggerisce di usare liste di stringhe, visto che non c'è necessità di spezzarle.