Problématique

Avant d'aborder ce chapitre il faut revoir le vocabulaire et les notions liés aux graphes ainsi que leur représentation en machine.

Problématique

Avant d'aborder ce chapitre il faut revoir le vocabulaire et les notions liés aux graphes ainsi que leur représentation en machine.

Etant donné un graphe (S, A), on s'intéresse de ce chapitre à des questions comme :

Problématique

Avant d'aborder ce chapitre il faut revoir le vocabulaire et les notions liés aux graphes ainsi que leur représentation en machine.

Etant donné un graphe (S, A), on s'intéresse de ce chapitre à des questions comme :

- Pour $(u, v) \in S^2$, existe-t-il un chemin de u à v ? Si oui, peut-on déterminer un chemin de plus faible coût ?

Problématique

Avant d'aborder ce chapitre il faut revoir le vocabulaire et les notions liés aux graphes ainsi que leur représentation en machine.

Etant donné un graphe (S, A), on s'intéresse de ce chapitre à des questions comme :

- Pour $(u, v) \in S^2$, existe-t-il un chemin de u à v ? Si oui, peut-on déterminer un chemin de plus faible coût ?
- Quels sont les sommets atteignables depuis un sommet donné ?

Problématique

Avant d'aborder ce chapitre il faut revoir le vocabulaire et les notions liés aux graphes ainsi que leur représentation en machine.

Etant donné un graphe (S, A), on s'intéresse de ce chapitre à des questions comme :

- Pour $(u, v) \in S^2$, existe-t-il un chemin de u à v ? Si oui, peut-on déterminer un chemin de plus faible coût ?
- Quels sont les sommets atteignables depuis un sommet donné ?
- G contient-il un cycle ?

Problématique

Avant d'aborder ce chapitre il faut revoir le vocabulaire et les notions liés aux graphes ainsi que leur représentation en machine.

Etant donné un graphe (S, A), on s'intéresse de ce chapitre à des questions comme :

- Pour $(u, v) \in S^2$, existe-t-il un chemin de u à v ? Si oui, peut-on déterminer un chemin de plus faible coût ?
- Quels sont les sommets atteignables depuis un sommet donné ?
- G contient-il un cycle ?
- ...

Problématique

Avant d'aborder ce chapitre il faut revoir le vocabulaire et les notions liés aux graphes ainsi que leur représentation en machine.

Etant donné un graphe (S, A), on s'intéresse de ce chapitre à des questions comme :

- Pour $(u, v) \in S^2$, existe-t-il un chemin de u à v ? Si oui, peut-on déterminer un chemin de plus faible coût ?
- Quels sont les sommets atteignables depuis un sommet donné ?
- G contient-il un cycle ?
- ...

Le langage d'implémentation des algorithmes sera prioritairement OCaml.

Rappel : itération sur les éléments d'une liste

- On rappelle qu'en OCaml, on peut appliquer une fonction aux éléments d'une liste avec `List.iter` de signature `('a -> unit) -> 'a list -> unit`, le premier argument est la fonction à appliquer et le second la liste des éléments. On notera bien que la fonction *renvoie* `unit`.

.

Exercices

Rappel : itération sur les éléments d'une liste

- On rappelle qu'en OCaml, on peut appliquer une fonction aux éléments d'une liste avec `List.iter` de signature `('a -> unit) -> 'a list -> unit`, le premier argument est la fonction à appliquer et le second la liste des éléments. On notera bien que la fonction *renvoie* `unit`.
  Par exemple, `List.iter print_int [1; 2; 3];;` affiche 123.

Exercices

Rappel : itération sur les éléments d'une liste

- On rappelle qu'en OCaml, on peut appliquer une fonction aux éléments d'une liste avec `List.iter` de signature `('a -> unit) -> 'a list -> unit`, le premier argument est la fonction à appliquer et le second la liste des éléments. On notera bien que la fonction *renvoie* `unit`.
  Par exemple, `List.iter print_int [1; 2; 3];;` affiche 123.
- La fonction à appliquer, peut être défini sous la forme d'une fonction anonyme avec `fun`.

Exercices

Rappel : itération sur les éléments d'une liste

- On rappelle qu'en OCaml, on peut appliquer une fonction aux éléments d'une liste avec `List.iter` de signature `('a -> unit) -> 'a list -> unit`, le premier argument est la fonction à appliquer et le second la liste des éléments. On notera bien que la fonction *renvoie* `unit`.
  Par exemple, `List.iter print_int [1; 2; 3];;` affiche 123.
- La fonction à appliquer, peut être défini sous la forme d'une fonction anonyme avec `fun`. Par exemple,
  ```
  List.iter (fun n -> print_int n; print_string "; ") [1; 2; 3];;
  ```
  affiche `1; 2; 3;`

Exercices

Rappel : itération sur les éléments d'une liste

- On rappelle qu'en OCaml, on peut appliquer une fonction aux éléments d'une liste avec `List.iter` de signature `('a -> unit) -> 'a list -> unit`, le premier argument est la fonction à appliquer et le second la liste des éléments. On notera bien que la fonction *renvoie* `unit`.
  Par exemple, `List.iter print_int [1; 2; 3];;` affiche 123.
- La fonction à appliquer, peut être défini sous la forme d'une fonction anonyme avec `fun`. Par exemple,
  ```
  List.iter (fun n -> print_int n; print_string "; ") [1; 2; 3];;
  ```
  affiche `1; 2; 3;`

Exercices

1. Utiliser `List.iter` pour afficher les longueurs d'une liste de chaîne de caractères.

Rappel : itération sur les éléments d'une liste

- On rappelle qu'en OCaml, on peut appliquer une fonction aux éléments d'une liste avec `List.iter` de signature `('a -> unit) -> 'a list -> unit`, le premier argument est la fonction à appliquer et le second la liste des éléments. On notera bien que la fonction *renvoie* `unit`.
  Par exemple, `List.iter print_int [1; 2; 3];;` affiche 123.
- La fonction à appliquer, peut être défini sous la forme d'une fonction anonyme avec `fun`. Par exemple,
  ```
  List.iter (fun n -> print_int n; print_string "; ") [1; 2; 3];;
  ```
  affiche `1; 2; 3;`

Exercices

1. Utiliser `List.iter` pour afficher les longueurs d'une liste de chaîne de caractères.
2. Ecrire, *en utilisant* `List.iter`, une fonction `somme: int list -> int` qui renvoie la somme des éléments de la liste d'entiers donnée en argument.

Parcours d'un graphe

A la base des algorithmes sur les graphes, on trouve les parcours de graphe, c'est à dire l'exploration des sommets. A partir du sommet de départ, on peut :

Parcours d'un graphe

A la base des algorithmes sur les graphes, on trouve les parcours de graphe, c'est à dire l'exploration des sommets. A partir du sommet de départ, on peut :

- explorer tous ses voisins immédiats, puis les voisins des voisins et ainsi de suite. Le graphe est donc exploré en « cercle concentrique »autour du sommet de départ . . ., on parle alors de parcours en largeur ou breadth first search (*BFS*) en anglais.

Parcours d'un graphe

A la base des algorithmes sur les graphes, on trouve les parcours de graphe, c'est à dire l'exploration des sommets. A partir du sommet de départ, on peut :

- explorer tous ses voisins immédiats, puis les voisins des voisins et ainsi de suite. Le graphe est donc exploré en « cercle concentrique »autour du sommet de départ . . ., on parle alors de parcours en largeur ou breadth first search (*BFS*) en anglais.
- explorer à chaque étape le premier voisin non encore exploré. Lorsque qu'on atteint un sommet dont tous les voisins ont déjà été exploré, on revient en arrière, on parle alors de parcours en profondeur ou depth first search (*DFS*) en anglais.

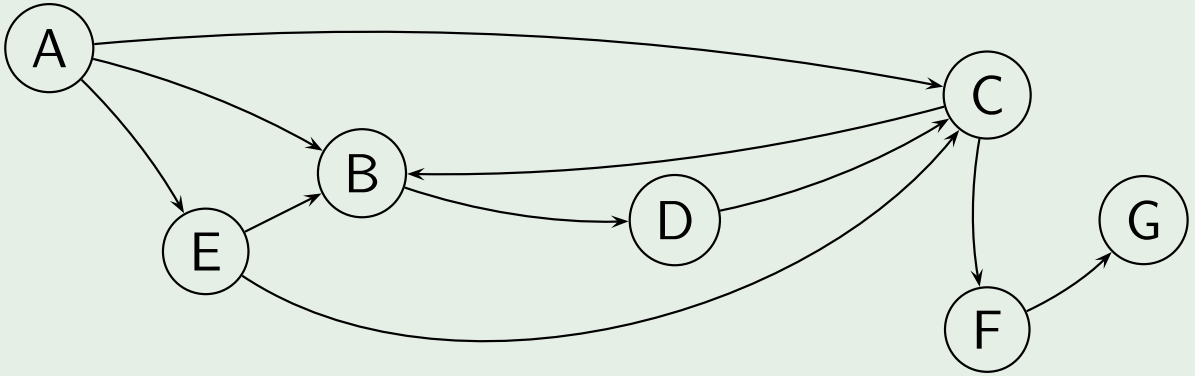
Parcours d'un graphe

A la base des algorithmes sur les graphes, on trouve les parcours de graphe, c'est à dire l'exploration des sommets. A partir du sommet de départ, on peut :

- explorer tous ses voisins immédiats, puis les voisins des voisins et ainsi de suite. Le graphe est donc exploré en « cercle concentrique »autour du sommet de départ . . ., on parle alors de parcours en largeur ou breadth first search (*BFS*) en anglais.
- explorer à chaque étape le premier voisin non encore exploré. Lorsque qu'on atteint un sommet dont tous les voisins ont déjà été exploré, on revient en arrière, on parle alors de parcours en profondeur ou depth first search (*DFS*) en anglais.

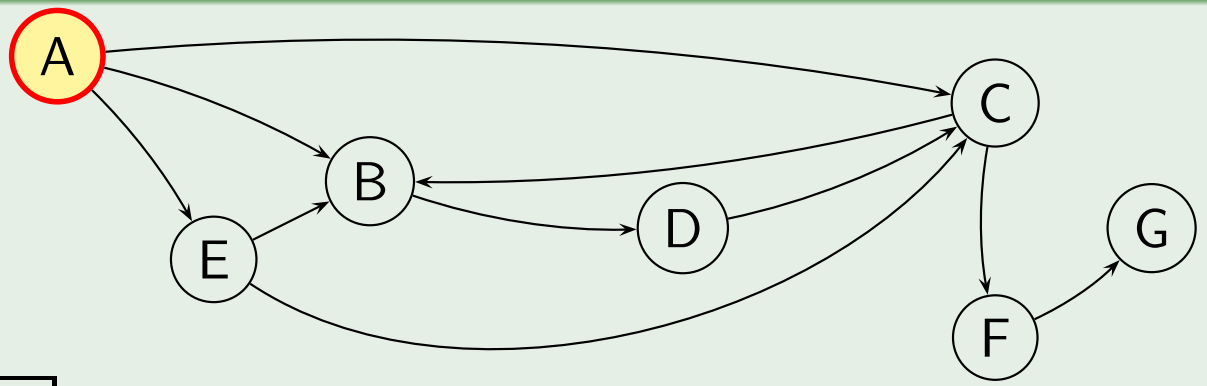
Ces deux parcours (`dfs` ou `bfs`) marquent exactement les sommets atteignables depuis la source (mais les sommets sont visités dans un ordre différent). Et les deux algorithmes peuvent donc être utilisés pour déterminer les *composantes connexes* d'un graphe.

Exemple de parcours en largeur



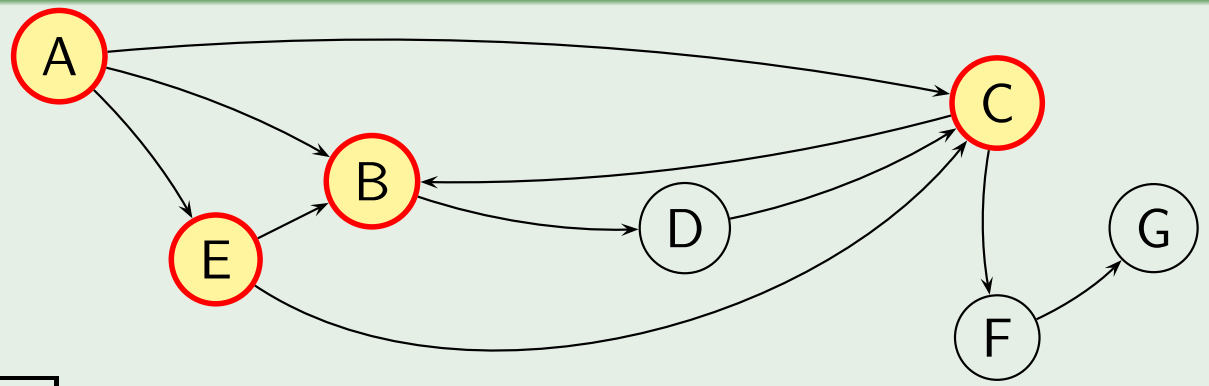
Sommets explorés :

Exemple de parcours en largeur



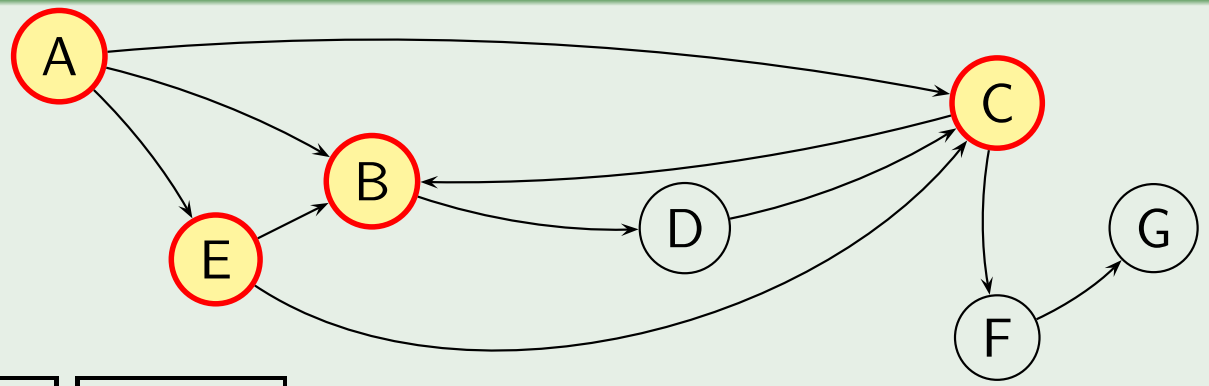
Sommets explorés : A,

Exemple de parcours en largeur



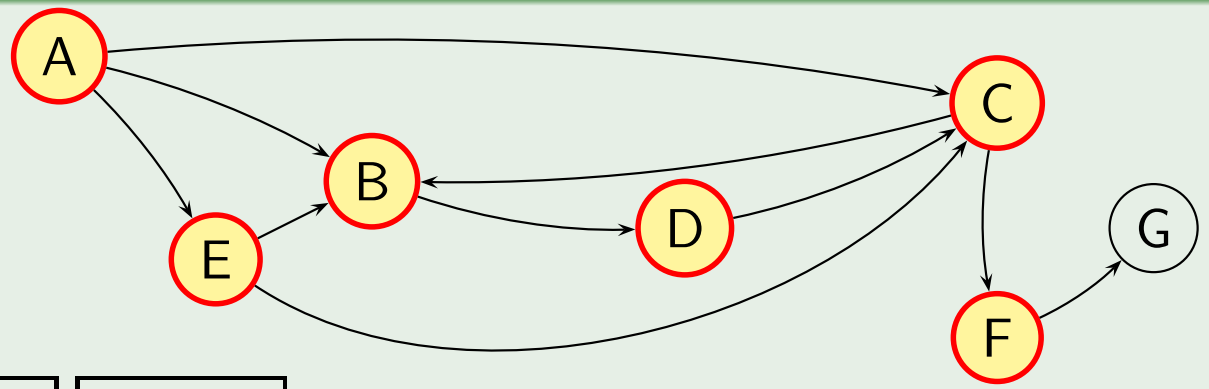
Sommets explorés : A,

Exemple de parcours en largeur



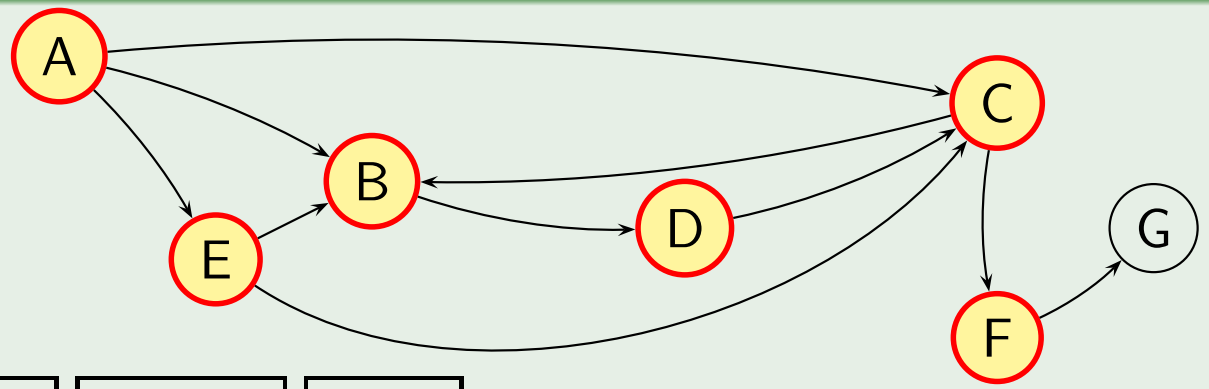
Sommets explorés : A, | B, C, E

Exemple de parcours en largeur



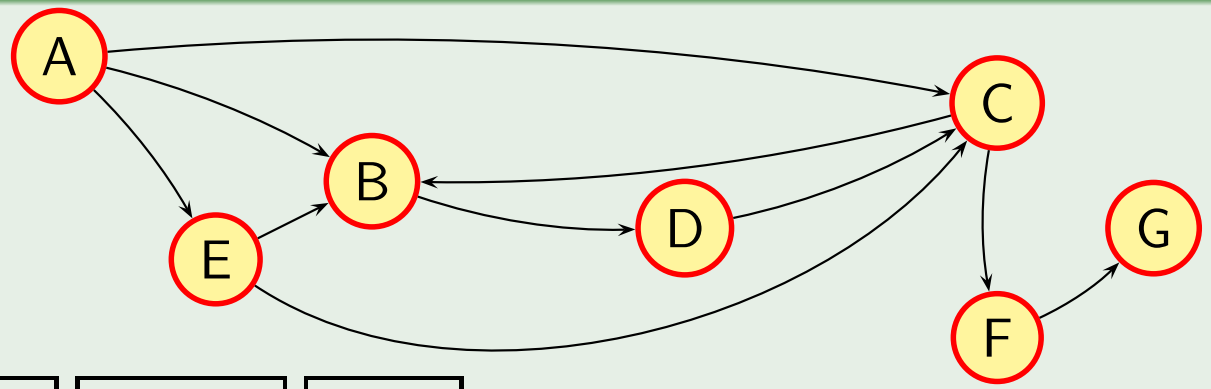
Sommets explorés : A, | B, C, E

Exemple de parcours en largeur



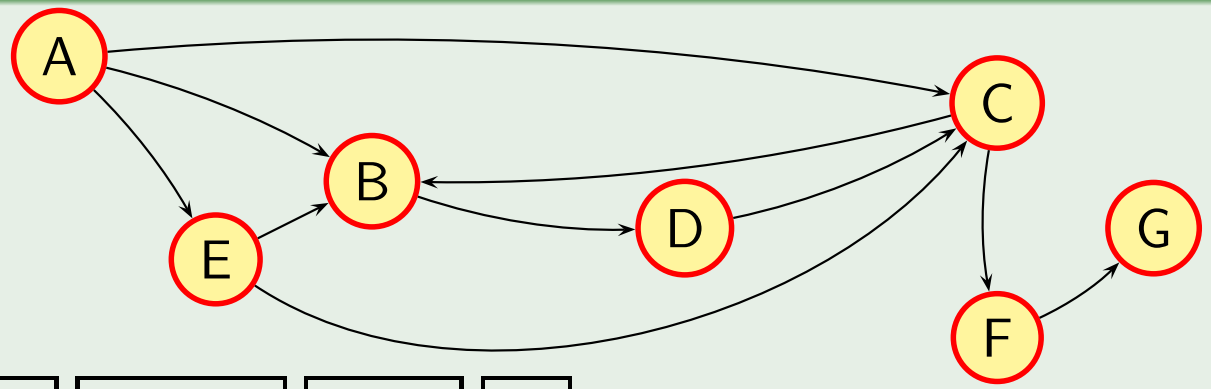
Sommets explorés : A, | B, C, E | D, F,

Exemple de parcours en largeur



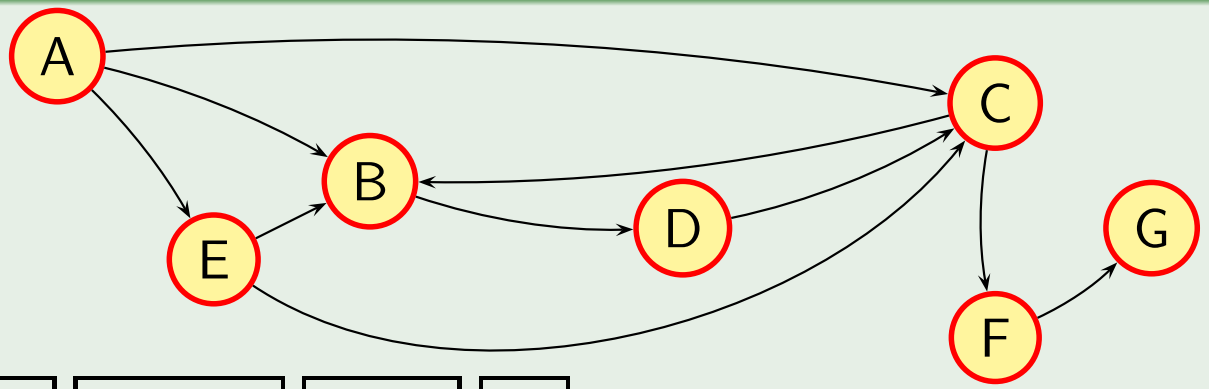
Sommets explorés : A, | B, C, E | D, F,

Exemple de parcours en largeur



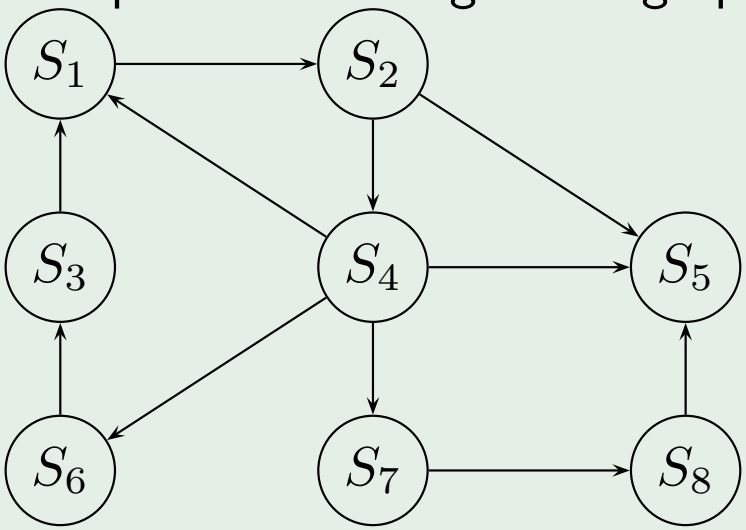
Sommets explorés : A, | B, C, E | D, F, | G

Exemple de parcours en largeur

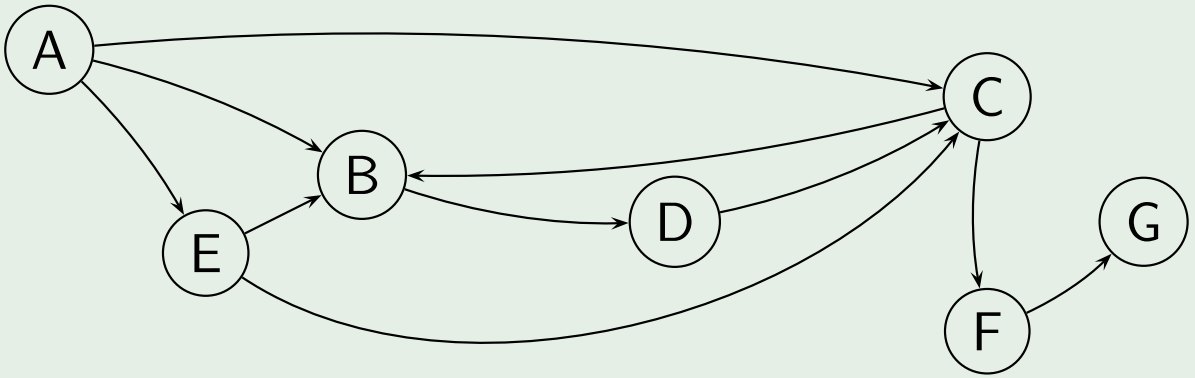


Sommets explorés : A, | B, C, E | D, F, | G

Ecrire le parcours en largeur du graphe suivant (on part du sommet S_1)

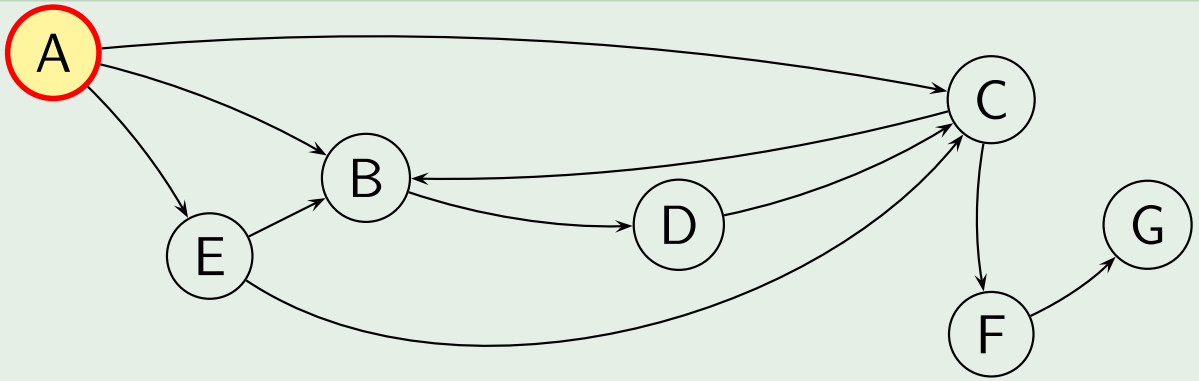


Exemple de parcours en profondeur



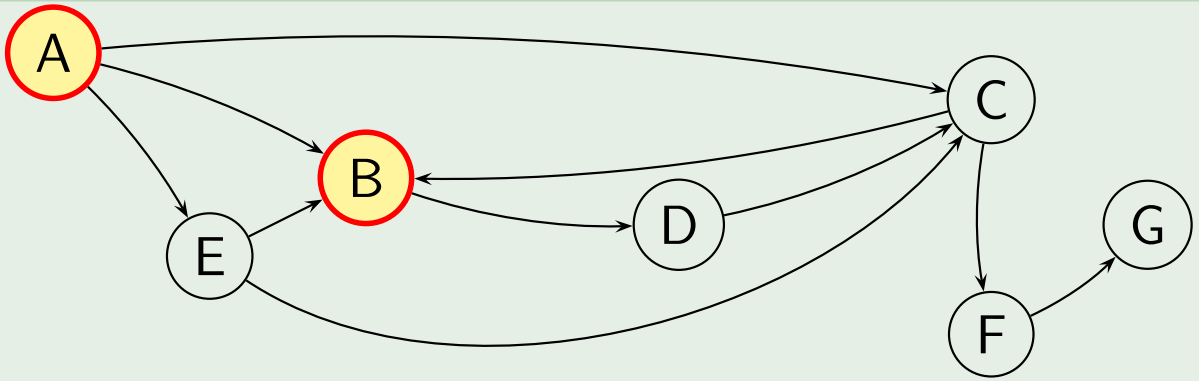
Sommets explorés :

Exemple de parcours en profondeur



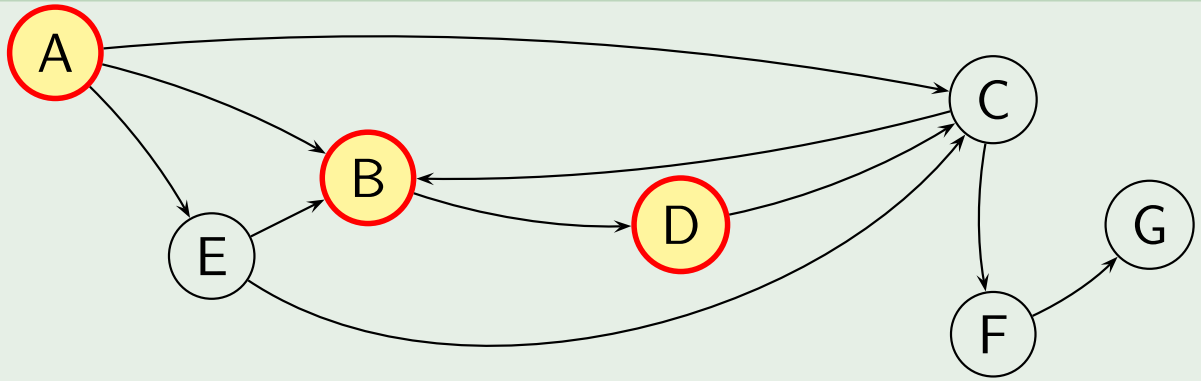
Sommets explorés : A,

Exemple de parcours en profondeur



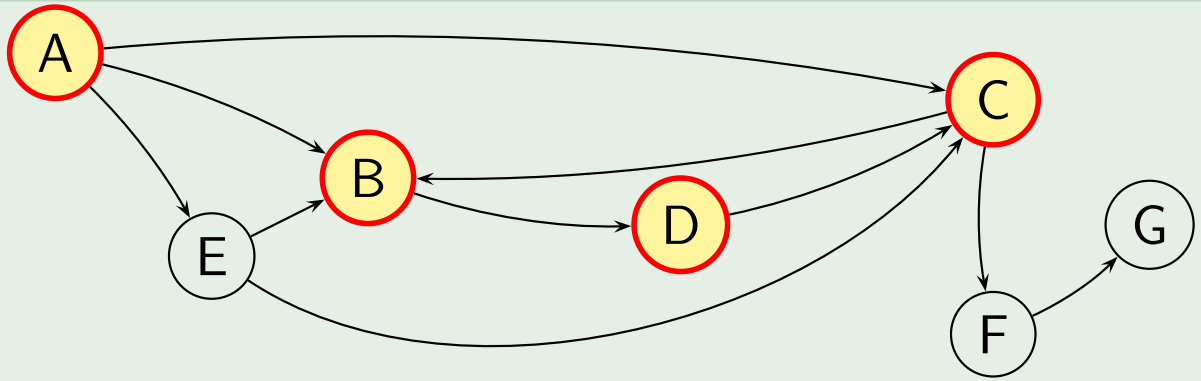
Sommets explorés : A, B,

Exemple de parcours en profondeur



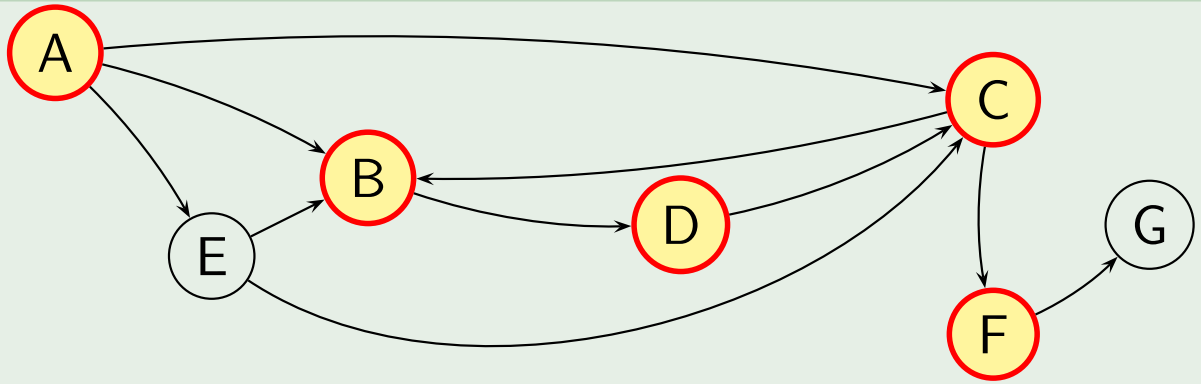
Sommets explorés : A, B, D,

Exemple de parcours en profondeur



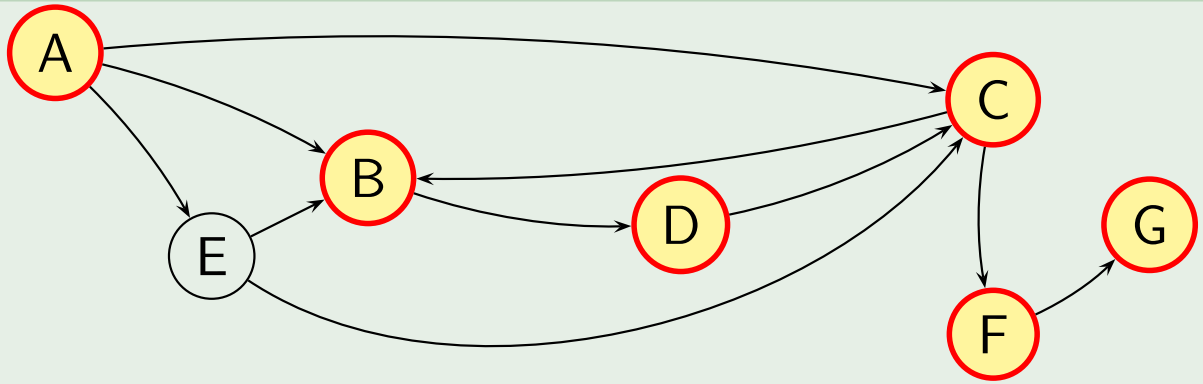
Sommets explorés : A, B, D, C,

Exemple de parcours en profondeur



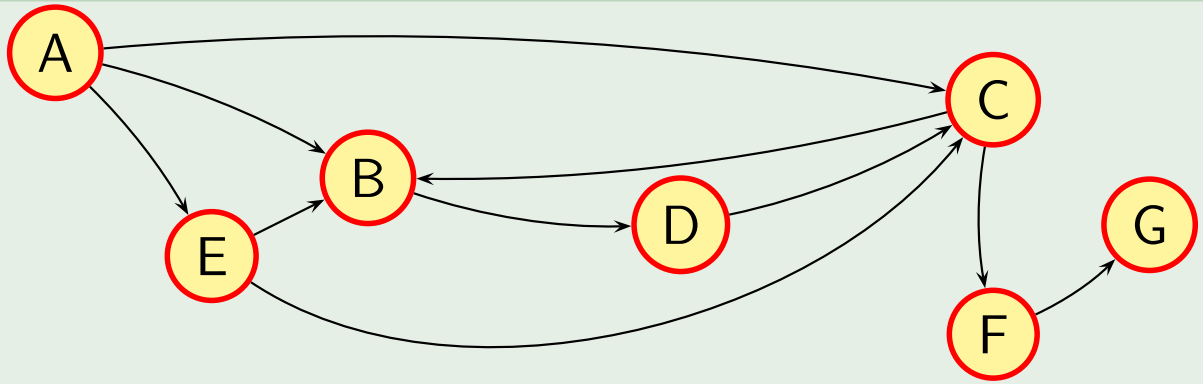
Sommets explorés : A, B, D, C, F,

Exemple de parcours en profondeur



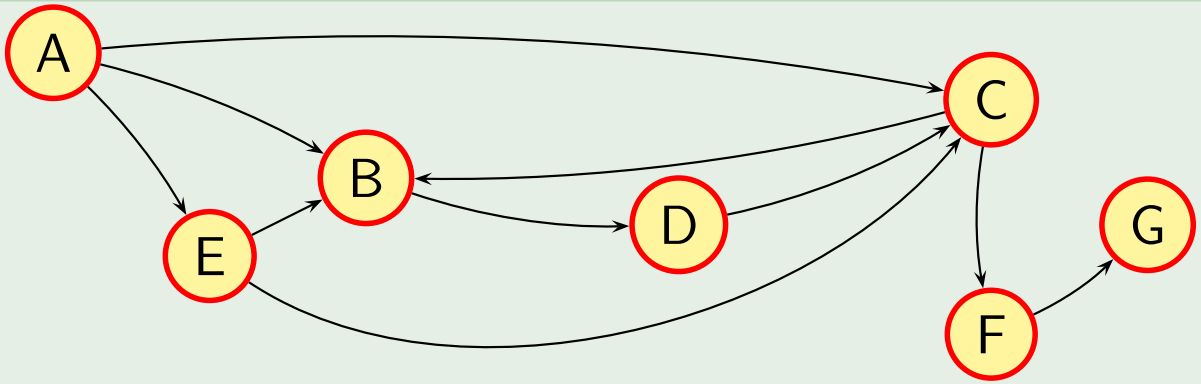
Sommets explorés : A, B, D, C, F, G,

Exemple de parcours en profondeur



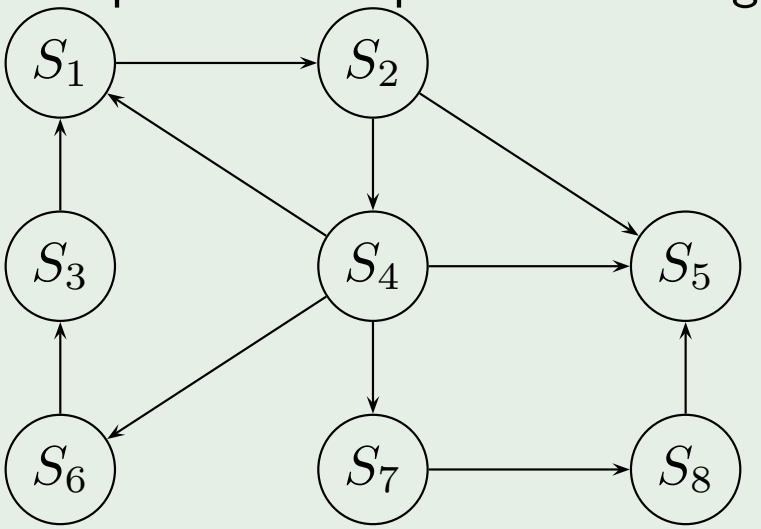
Sommets explorés : A, B, D, C, F, G, E

Exemple de parcours en profondeur



Sommets explorés : A, B, D, C, F, G, E

Ecrire le parcours en profondeur du graphe suivant (on part du sommet S_1)



File et parcours en largeur

- Pour un parcours en largeur, on doit stocker dans une structure de données les sommets en attente d'être explorés. C'est à dire les voisins du sommet de départ, puis les voisins des voisins . . .Ces sommets doivent être retirés pour exploration, dans leur ordre d'insertion, la structure de données utilisée est donc du type premier entré, premier sorti (first in first out (*FIFO*)), c'est donc une file qu'on implémentera à l'aide du module `Queue` de OCaml.

File et parcours en largeur

- Pour un parcours en largeur, on doit stocker dans une structure de données les sommets en attente d'être explorés. C'est à dire les voisins du sommet de départ, puis les voisins des voisins . . .Ces sommets doivent être retirés pour exploration, dans leur ordre d'insertion, la structure de données utilisée est donc du type premier entré, premier sorti (first in first out (*FIFO*)), c'est donc une file qu'on implémentera à l'aide du module `Queue` de OCaml.
- L'algorithme de parcours en largeur (BFS) est donc le suivant :

File et parcours en largeur

- Pour un parcours en largeur, on doit stocker dans une structure de données les sommets en attente d'être explorés. C'est à dire les voisins du sommet de départ, puis les voisins des voisins . . .Ces sommets doivent être retirés pour exploration, dans leur ordre d'insertion, la structure de données utilisée est donc du type premier entré, premier sorti (first in first out (*FIFO*)), c'est donc une file qu'on implémentera à l'aide du module `Queue` de OCaml.
- L'algorithme de parcours en largeur (BFS) est donc le suivant :
  - Initialement, la file contient le noeud de départ

File et parcours en largeur

- Pour un parcours en largeur, on doit stocker dans une structure de données les sommets en attente d'être explorés. C'est à dire les voisins du sommet de départ, puis les voisins des voisins . . .Ces sommets doivent être retirés pour exploration, dans leur ordre d'insertion, la structure de données utilisée est donc du type premier entré, premier sorti (first in first out (*FIFO*)), c'est donc une file qu'on implémentera à l'aide du module `Queue` de OCaml.
- L'algorithme de parcours en largeur (BFS) est donc le suivant :
  - Initialement, la file contient le noeud de départ
  - A chaque étape, on défile un noeud, *on marque ce noeud comme visité*, et on enfile ses fils *non encore marqués*.

Implémentation en OCaml

On utilise la représentation par liste d'adjacence des graphes orientés et on rappelle que dans ce cas, les listes d'adjacence contiennent les voisins sortants :

```
1  type digraph = {taille : int;  ladj : int list array};;
```

On rappelle les fonctions principales du module `Queue` :

- `Queue.create: unit -> 'a Queue` qui crée une file vide d'éléments de type 'a.
- `Queue.push: 'a Queue -> 'a -> unit` qui enfile un élément.
- `Queue.pop 'a Queue -> 'a` qui défile.
- `Queue.is_empty 'a Queue -> bool` qui test si la file est vide.

Exercice

Exercice

1. Ecrire `cree_graphe : int -> digraph` qui prend en argument un entier `n` et renvoie un graphe orienté vide de taille `n`.

Exercice

1. Ecrire `cree_graphe: int -> digraph` qui prend en argument un entier `n` et renvoie un graphe orienté vide de taille `n`.
2. Ecrire `ajoute_aretes: digraphe -> (int*int) list -> unit` qui prend en argument une liste d'arêtes, chaque arête étant une couple d'entiers (somme de départ, sommet d'arrivée) et ajoute ces arêtes dans le graphe donné en argument.

Exercice

1. Ecrire `cree_graphe: int -> digraph` qui prend en argument un entier `n` et renvoie un graphe orienté vide de taille `n`.
2. Ecrire `ajoute_aretes: digraphe -> (int*int) list -> unit` qui prend en argument une liste d'arêtes, chaque arête étant une couple d'entiers (somme de départ, sommet d'arrivée) et ajoute ces arêtes dans le graphe donné en argument.
3. Ecrire la fonction `bfs: digraphe -> int -> int array` qui renvoie le résultat du parcours en largeur d'un graphe sous la forme d'un tableau de distance (en nombre d'arêtes depuis le sommet de départ).
   - On pourra procéder avec les étapes suivantes :
     - créer une file vide et y enfiler le sommet de départ
     - créer le tableau de distance en initialisant à -1 (ou `Int.max_int`) pour indiquer que le sommet n'a pas encore été visité sauf pour le sommet de départ dont la distance est 0.
     - Ecrire la boucle principale : tant que la file n'est pas vide, à chaque étape on extrait un sommet et on enfile ses fils *n'ayant pas encore été parcourus*

Proposition de correction

```
let bfs graphe depart =
  let a_traiter = Queue.create() in
  let distance = Array.make graphe.taille (-1) in
  distance.(depart) <- 0;
  Queue.push  depart a_traiter;
  while not (Queue.is_empty a_traiter) do
    let sommet_courant = Queue.pop a_traiter in
    List.iter (fun s -> if (distance.(s)= -1) then
     ↪  (Queue.push s a_traiter;
     ↪  distance.(s)<-distance.(sommet_courant) + 1))
     ↪  (graphe.ladj.(sommet_courant));
  done;
  distance;;
```

Analyse de l'algorithme BFS

Analyse de l'algorithme BFS

- Chaque sommet est inséré une unique fois dans la file et à chaque tour de boucle on retire un sommet ce qui assure la terminaison

Analyse de l'algorithme BFS

- Chaque sommet est inséré une unique fois dans la file et à chaque tour de boucle on retire un sommet ce qui assure la terminaison
- Pour la correction, on prouve l'invariant suivant : pour le sommet `s` retiré de la file à chaque tour de boucle, `distance.(s)` contient la distance minimale (en nombre d'arêtes) du sommet de départ à `s`.

Analyse de l'algorithme BFS

- Chaque sommet est inséré une unique fois dans la file et à chaque tour de boucle on retire un sommet ce qui assure la terminaison
- Pour la correction, on prouve l'invariant suivant : pour le sommet `s` retiré de la file à chaque tour de boucle, `distance.(s)` contient la distance minimale (en nombre d'arêtes) du sommet de départ à `s`.
  - C'est vrai au premier tour de boucle puisque le premier sommet retiré est le sommet de départ et qu'on initialise `distance.(depart)` à 0.

Analyse de l'algorithme BFS

- Chaque sommet est inséré une unique fois dans la file et à chaque tour de boucle on retire un sommet ce qui assure la terminaison
- Pour la correction, on prouve l'invariant suivant : pour le sommet `s` retiré de la file à chaque tour de boucle, `distance.(s)` contient la distance minimale (en nombre d'arêtes) du sommet de départ à `s`.
  - C'est vrai au premier tour de boucle puisque le premier sommet retiré est le sommet de départ et qu'on initialise `distance.(depart)` à 0.
  - La propriété est conservée à chaque tour de boucle, en effet en la supposant vraie à tous les tours de boucle précédents, les sommets présents dans la file sont les sommets `t` non encore parcourus et tels que `distance.(t)` contienne `distance.(t) = distance.(s) + 1` où `s` est un sommet déjà parcouru. Puisque `s` a déjà été retiré de la file `distance.(s)` est minimale et donc `distance.(t)` aussi.

Analyse de l'algorithme BFS

- Chaque sommet est inséré une unique fois dans la file et à chaque tour de boucle on retire un sommet ce qui assure la terminaison
- Pour la correction, on prouve l'invariant suivant : pour le sommet `s` retiré de la file à chaque tour de boucle, `distance.(s)` contient la distance minimale (en nombre d'arêtes) du sommet de départ à `s`.
  - C'est vrai au premier tour de boucle puisque le premier sommet retiré est le sommet de départ et qu'on initialise `distance.(depart)` à 0.
  - La propriété est conservée à chaque tour de boucle, en effet en la supposant vraie à tous les tours de boucle précédents, les sommets présents dans la file sont les sommets `t` non encore parcourus et tels que `distance.(t)` contienne `distance.(t) = distance.(s) + 1` où `s` est un sommet déjà parcouru. Puisque `s` a déjà été retiré de la file `distance.(s)` est minimale et donc `distance.(t) aussi`.
- La boucle `while` s'exécute au plus $|S|$ fois et l'itération sur les voisins sortants d'un sommet permet d'examiner chaque arête au plus une fois. La complexité du parcours en largeur est donc un $O(|S| + |A|)$

Pile et parcours en profondeur

- Pour un parcours en profondeur, on stocke aussi dans une structure de données les sommets en attente d'être explorés. Mais cette fois, la structure de données utilisée est du type dernier entré, premier sorti (last in first out (*LIFO*)), c'est à dire une pile.

Pile et parcours en profondeur

- Pour un parcours en profondeur, on stocke aussi dans une structure de données les sommets en attente d'être explorés. Mais cette fois, la structure de données utilisée est du type dernier entré, premier sorti (last in first out (*LIFO*)), c'est à dire une pile.

Pile et parcours en profondeur

- Pour un parcours en profondeur, on stocke aussi dans une structure de données les sommets en attente d'être explorés. Mais cette fois, la structure de données utilisée est du type dernier entré, premier sorti (last in first out (*LIFO*)), c'est à dire une pile.
- Pour l'implémentation, on peut :

Pile et parcours en profondeur

- Pour un parcours en profondeur, on stocke aussi dans une structure de données les sommets en attente d'être explorés. Mais cette fois, la structure de données utilisée est du type dernier entré, premier sorti (last in first out (*LIFO*)), c'est à dire une pile.
- Pour l'implémentation, on peut :
  - se contenter d'utiliser la récursivité de façon à ce que la pile des sommets en attente d'être exploré soit gérée de façon automatique via la pile des appels récursifs.

Pile et parcours en profondeur

- Pour un parcours en profondeur, on stocke aussi dans une structure de données les sommets en attente d'être explorés. Mais cette fois, la structure de données utilisée est du type dernier entré, premier sorti (last in first out (*LIFO*)), c'est à dire une pile.
- Pour l'implémentation, on peut :
  - se contenter d'utiliser la récursivité de façon à ce que la pile des sommets en attente d'être exploré soit gérée de façon automatique via la pile des appels récursifs.
  - Utiliser le module `Stack` de OCaml afin de gérer directement la pile des sommets en attente d'être exploré.

Pile et parcours en profondeur

- Pour un parcours en profondeur, on stocke aussi dans une structure de données les sommets en attente d'être explorés. Mais cette fois, la structure de données utilisée est du type dernier entré, premier sorti (last in first out (*LIFO*)), c'est à dire une pile.
- Pour l'implémentation, on peut :
  - se contenter d'utiliser la récursivité de façon à ce que la pile des sommets en attente d'être exploré soit gérée de façon automatique via la pile des appels récursifs.
  - Utiliser le module `Stack` de OCaml afin de gérer directement la pile des sommets en attente d'être exploré.

⚠ La différence entre les deux parcours ne se limite pas à l'utilisation d'une pile ou d'une file. En effet, dans le cas d'un parcours en largeur, un sommet ne peut jamais être inséré deux fois dans la file. Dans le parcours d'un parcours en profondeur, on peut effectuer un appel récursif sur le même sommet plusieurs fois (ou un sommet peut apparaître plusieurs fois dans la pile) mais la seconde fois, l'appel récursif se termine immédiatement.

Implémentation en OCaml

Comme pour le parcours en largeur, On utilise la représentation par liste d'adjacence des graphes orientés :

```
1  type digraph = {taille : int;  ladj : int list array};;
```

Implémentation en OCaml

Comme pour le parcours en largeur, On utilise la représentation par liste d'adjacence des graphes orientés :

```
1  type digraph = {taille : int;  ladj : int list array};;
```

On utilise ici la pile d'appel récursif *en gardant la trace des sommets visités*. On renvoie donc un tableau de booléens `visite` qui vaut `true` si le sommet a été rencontré lors du parcours et `false sinon`.

Implémentation en OCaml

Comme pour le parcours en largeur, On utilise la représentation par liste d'adjacence des graphes orientés :

```
1  type digraph = {taille : int;  ladj : int list array};;
```

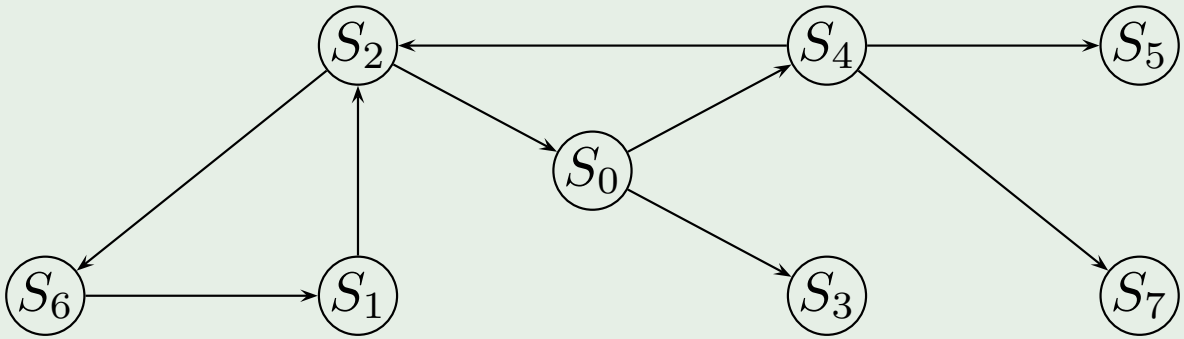
On utilise ici la pile d'appel récursif *en gardant la trace des sommets visités*. On renvoie donc un tableau de booléens `visite` qui vaut `true` si le sommet a été rencontré lors du parcours et `false sinon`. L'algorithme consiste à relancer récursivement le parcours sur les fils du noeud courant non encore visités. Un appel récursif sur un noeud déjà visité se termine immédiatement.

Parcours en profondeur

```
1  let dfs graphe depart =
2    (* Le tableau de booléens indiquant les sommets visités*)
3    let deja_vu =  Array.make graphe.taille false in
4    let rec aux_dfs s =
5      (* On relance récursivement sur les voisins sortants *)
6      if (not deja_vu.(s)) then (
7        deja_vu.(s) <- true;
8        List.iter aux_dfs (graphe.ladj.(s));) in
9    (* L'appel récursif termine immédiatement sur un voisin
     ↪  déjà visité *)
10   aux_dfs depart;
11   deja_vu;;
```

Exercice

Dérouler soigneusement l'algorithme `dfs` sur le graphe ci-dessous en partant du sommet S_0 et en indiquant à chaque fois le sommet sur lequel la fonction auxiliaire `aux_dfs` est appelée. On supposera que les voisins sont traités dans l'ordre croissant.



Exercice

Ecrire la fonction `dfs_stack: digraphe -> int -> boolarray` de parcours en profondeur en version itérative, on utilise donc une pile (module `Stack` de OCaml) pour stocker les sommets en attente de traitement. On rappelle les fonctions principales de ce module :

- `Stack.create` de signature `() -> 'a t` qui crée une pile vide d'éléments de type `'a`. Par exemple `let mapile = Stack.create ()`
- `Stack.push` de signature `'a -> 'a t -> ()` qui empile un élément. Par exemple `Stack.push 5 mapile` empile l'entier 5 sur `mapile` (le type option 'a est alors le type `int`).
- `Stack.pop` de signature `'a t -> 'a` qui renvoie l'élément situé au sommet de la pile en le dépilant.

S'inspirer de la version récursive !

Problème d'ordonnancement

Un graphe orienté permet de modéliser un problème d'ordonnancement :

- Un sommet est une tâche à effectuer

Problème d'ordonnancement

Un graphe orienté permet de modéliser un problème d'ordonnancement :

- Un sommet est une tâche à effectuer
- Un arc $s \to t$, indique que la tache s doit être effectuée *avant* la tâche t.

Problème d'ordonnancement

Un graphe orienté permet de modéliser un problème d'ordonnancement :

- Un sommet est une tâche à effectuer
- Un arc $s \to t$, indique que la tache s doit être effectuée *avant* la tâche t.

Par exemple, si le graphe représente les tâches nécessaires à la construction d'une maison, on doit faire la tâche « peindre les murs » *après* la tache « monter les murs ».

Problème d'ordonnancement

Un graphe orienté permet de modéliser un problème d'ordonnancement :

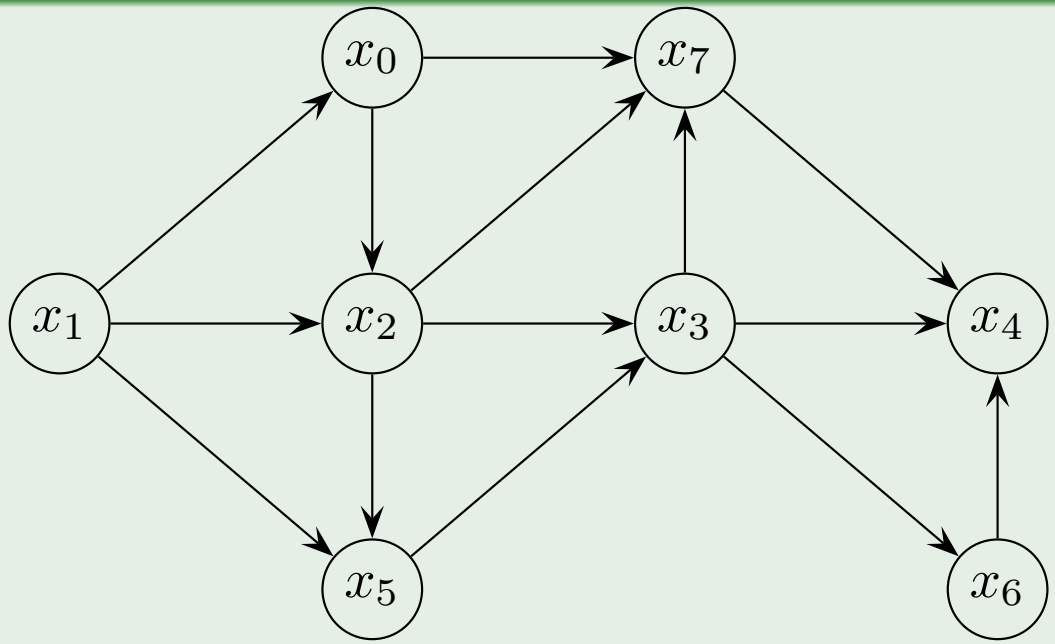
- Un sommet est une tâche à effectuer
- Un arc $s \to t$, indique que la tache s doit être effectuée *avant* la tâche t.

Par exemple, si le graphe représente les tâches nécessaires à la construction d'une maison, on doit faire la tâche « peindre les murs » *après* la tache « monter les murs ».

Définition

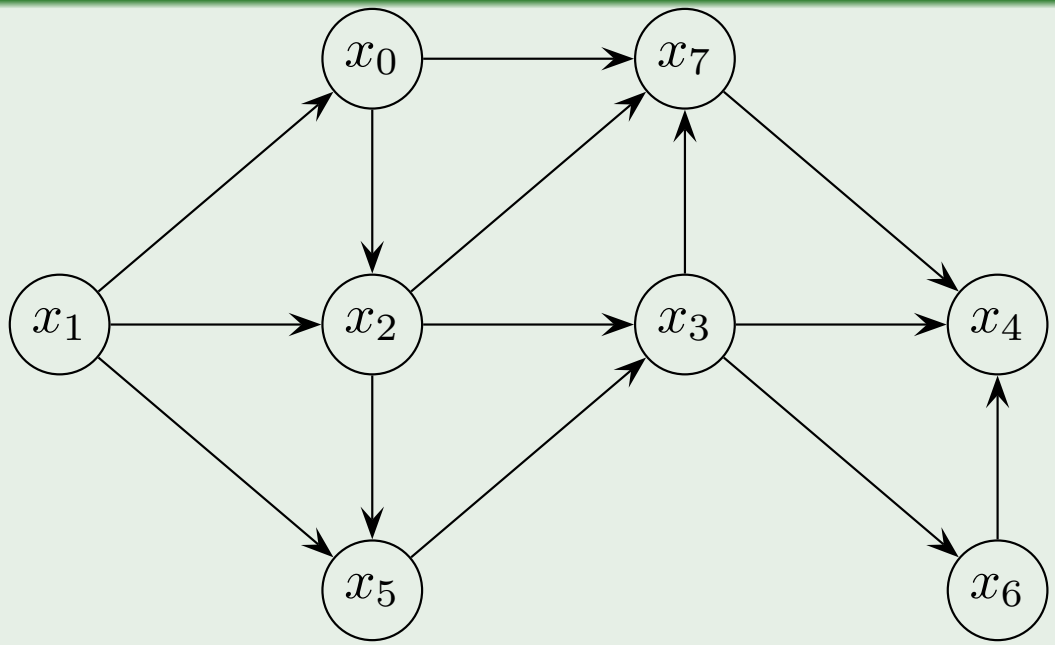
Le tri topologique d'un graphe orienté acyclique est une liste ordonnée de ses sommets tels que pour tout arc $u \to v$, le sommet u apparaît avant le sommet v dans la liste.

Exemple



1. Parmi les listes suivantes laquelle est un tri topologique ?
   - $L_1 = \{x_1, x_2, x_0, x_5, x_7, x_3, x_4, x_6\}$
   - $L_2 = \{x_1, x_0, x_2, x_5, x_3, x_6, x_4, x_7\}$
   - $L_3 = \{x_1, x_0, x_2, x_5, x_3, x_6, x_7, x_4\}$
   - $L_4 = \{x_1, x_0, x_2, x_3, x_4, x_5, x_6, x_7\}$

Exemple



1. Parmi les listes suivantes laquelle est un tri topologique ?
   - $L_1 = \{x_1, x_2, x_0, x_5, x_7, x_3, x_4, x_6\}$
   - $L_2 = \{x_1, x_0, x_2, x_5, x_3, x_6, x_4, x_7\}$
   - $L_3 = \{x_1, x_0, x_2, x_5, x_3, x_6, x_7, x_4\}$
   - $L_4 = \{x_1, x_0, x_2, x_3, x_4, x_5, x_6, x_7\}$
2. Déterminer (si possible) un autre tri topologique pour ce graphe.

Ordre postfixe d'un parcours en profondeur

On peut modifier le parcours en profondeur afin de garder la trace dans une liste de l'ordre dans lequel les sommets ont été visités.

Ordre postfixe d'un parcours en profondeur

On peut modifier le parcours en profondeur afin de garder la trace dans une liste de l'ordre dans lequel les sommets ont été visités.

- Pour garantir la visite de tous les sommets, on doit effectuer un parcours en profondeur à partir de *chaque sommet du graphe*.

Ordre postfixe d'un parcours en profondeur

On peut modifier le parcours en profondeur afin de garder la trace dans une liste de l'ordre dans lequel les sommets ont été visités.

- Pour garantir la visite de tous les sommets, on doit effectuer un parcours en profondeur à partir de *chaque sommet du graphe*.
- On ajoute un sommet *après* avoir relancé le parcours sur tous ces fils

Ordre postfixe d'un parcours en profondeur

On peut modifier le parcours en profondeur afin de garder la trace dans une liste de l'ordre dans lequel les sommets ont été visités.

- Pour garantir la visite de tous les sommets, on doit effectuer un parcours en profondeur à partir de *chaque sommet du graphe*.
- On ajoute un sommet *après* avoir relancé le parcours sur tous ces fils

On obtient ainsi l'ordre postfixe d'un graphe orienté acyclique.

Ordre postfixe d'un parcours en profondeur

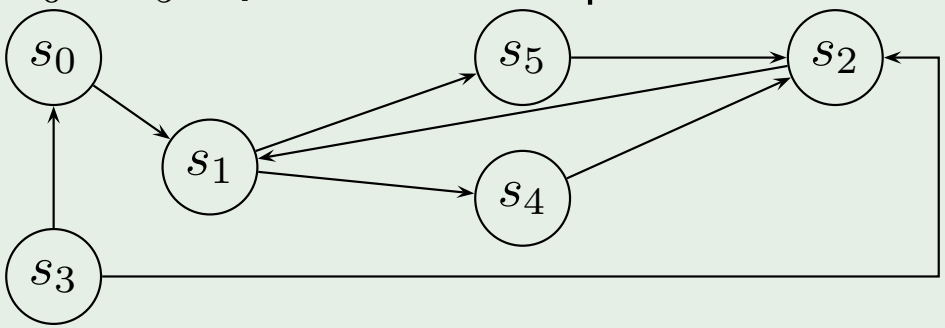
On peut modifier le parcours en profondeur afin de garder la trace dans une liste de l'ordre dans lequel les sommets ont été visités.

- Pour garantir la visite de tous les sommets, on doit effectuer un parcours en profondeur à partir de *chaque sommet du graphe*.
- On ajoute un sommet *après* avoir relancé le parcours sur tous ces fils

On obtient ainsi l'ordre postfixe d'un graphe orienté acyclique.

Exemple

Dérouler soigneusement l'algorithme `bfs` sur le graphe suivant *pour chaque sommet* de s_0 à s_5. Quel est l'ordre postfixe obtenu ?



Propriété

Sur un graphe orienté acyclique, l'ordre postfixe est un tri topologique.

Propriété

Sur un graphe orienté acyclique, l'ordre postfixe est un tri topologique.

Pour la preuve, on considère $u \to v \in A$, et on montre par disjonction de cas que `dfs u` termine *après* `dfs v`

Propriété

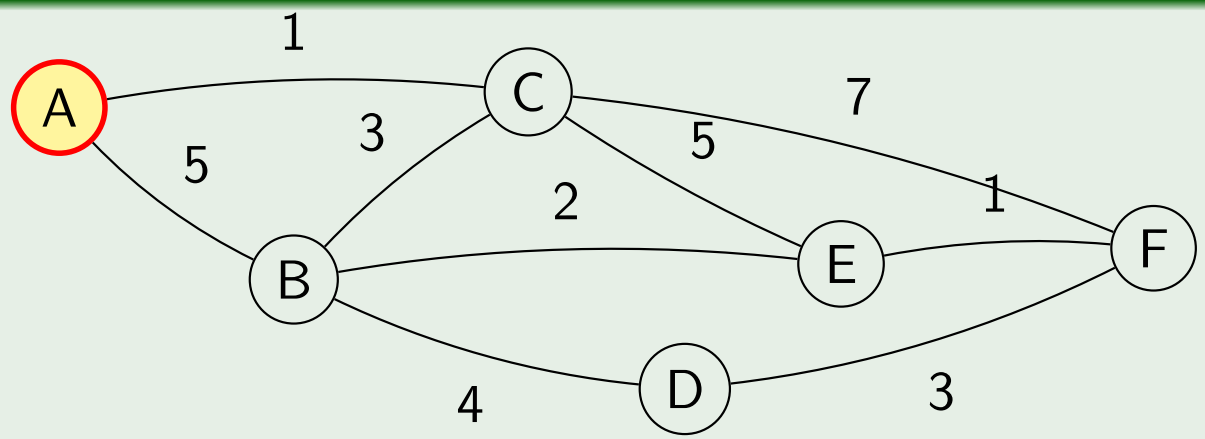
Sur un graphe orienté acyclique, l'ordre postfixe est un tri topologique.

Pour la preuve, on considère $u \to v \in A$, et on montre par disjonction de cas que `dfs u` termine *après* `dfs v`

Implémentation en OCaml

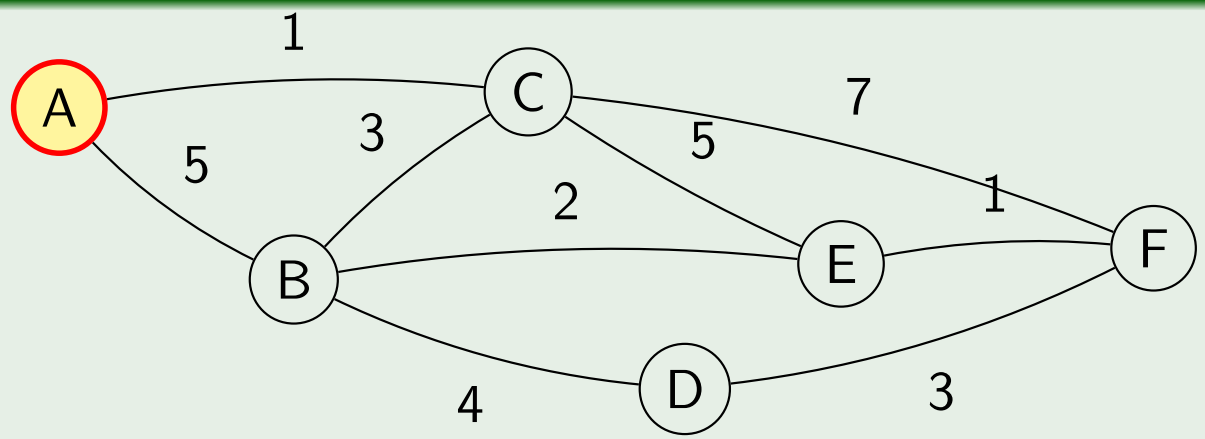
```
1      let ordre_dfs graphe =
2       let deja_vu =  Array.make graphe.taille false in
3       let ordre = ref [] in
4       let rec aux_dfs s =
5         if (not deja_vu.(s)) then (
6           deja_vu.(s) <- true;
7           List.iter aux_dfs (graphe.ladj.(s));
8           ordre := s::(!ordre);
9           ) in
10      List.iter aux_dfs (List.init (graphe.taille) (fun i-> i));
11     !ordre;;
```

Algorithme de Djikstra : exemple



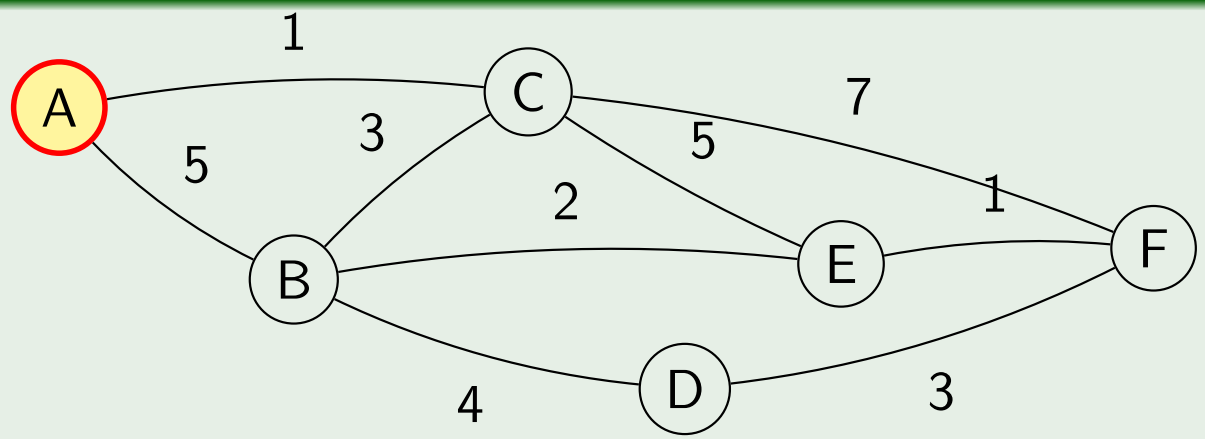
A	B	C	D	E	F	

Algorithme de Djikstra : exemple



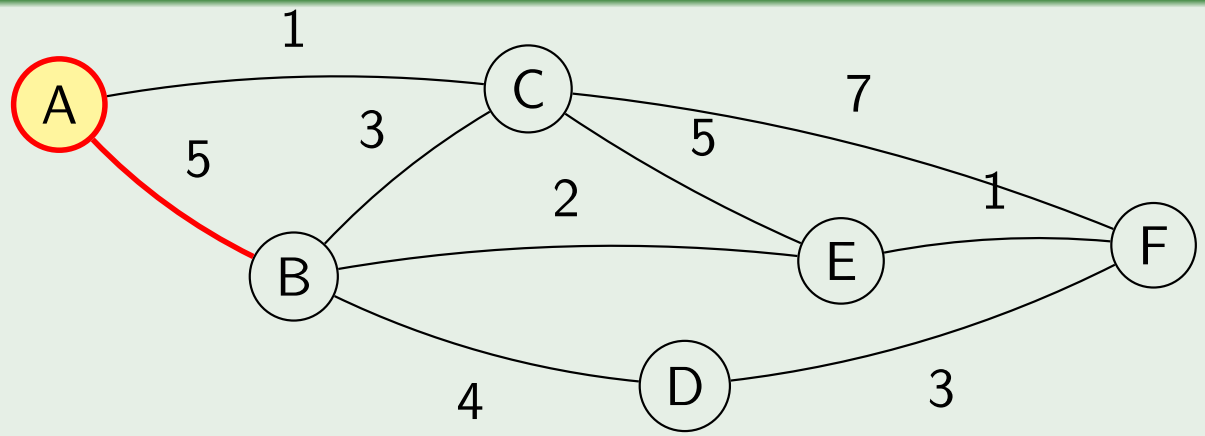
A	B	C	D	E	F	
0 (A)						

Algorithme de Djikstra : exemple



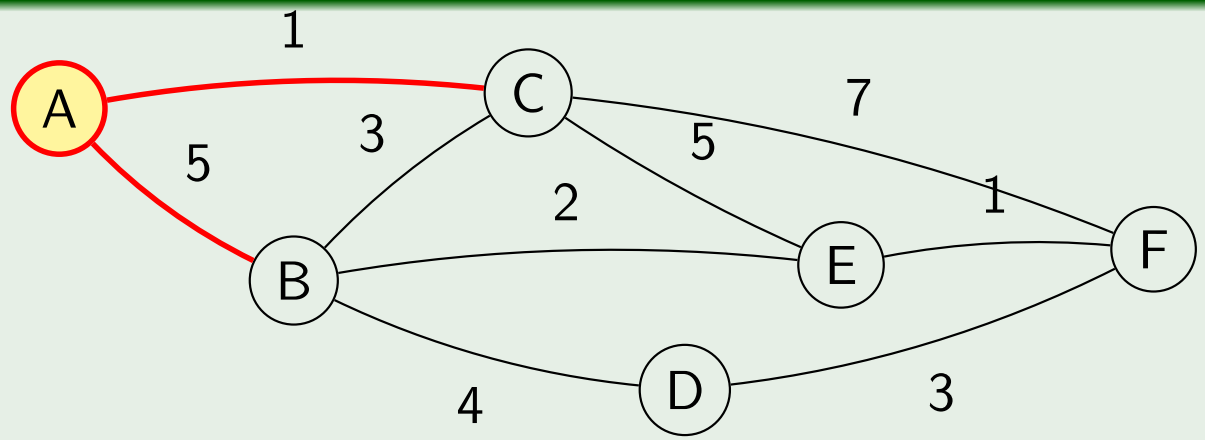
A	B	C	D	E	F	
0 (A)						A
✔						
✔						
✔						
✔						
✔						

Algorithme de Djikstra : exemple



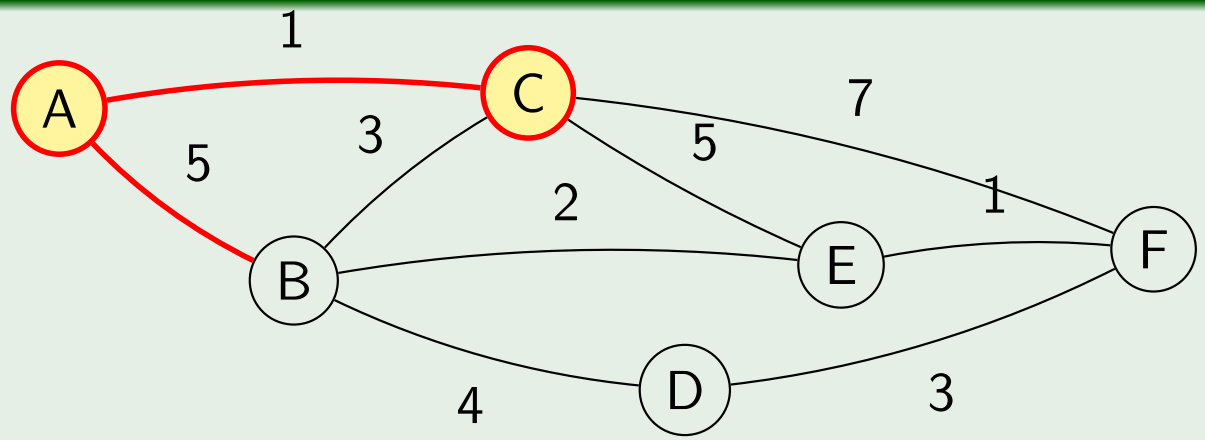
A	B	C	D	E	F	
0 (A)	5 (A)					A
✔						
✔						
✔						
✔						
✔						

Algorithme de Djikstra : exemple



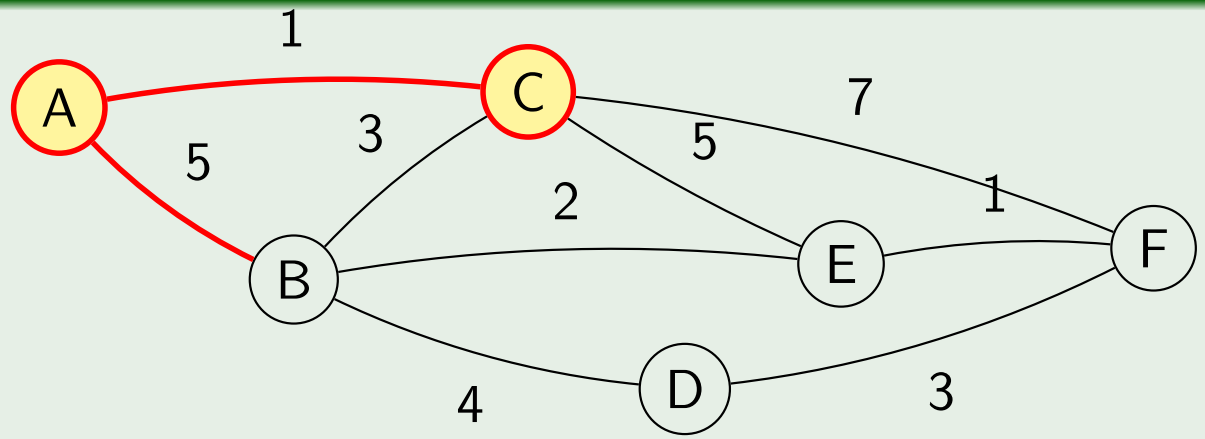
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔						
✔						
✔						
✔						
✔						

Algorithme de Djikstra : exemple



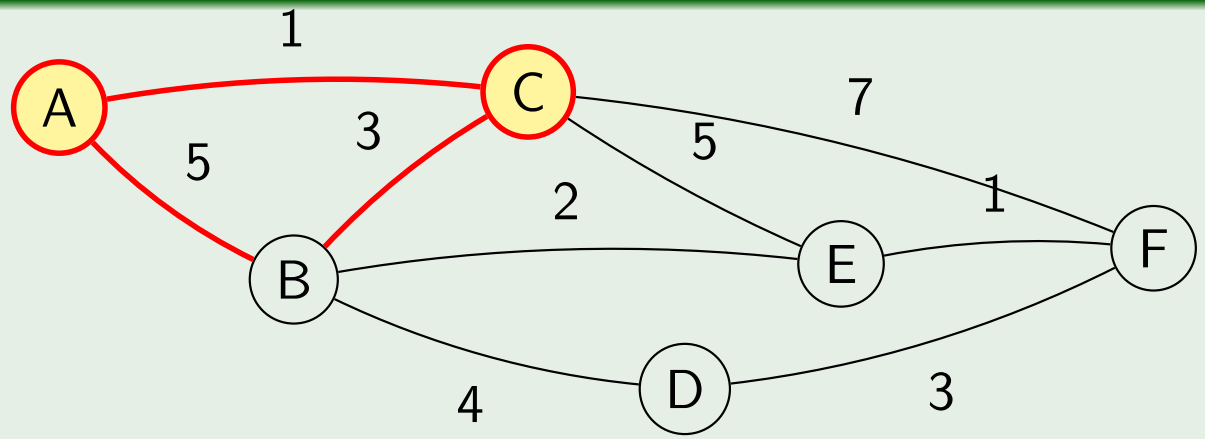
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔		1 (A)				
✔						
✔						
✔						
✔						

Algorithme de Djikstra : exemple



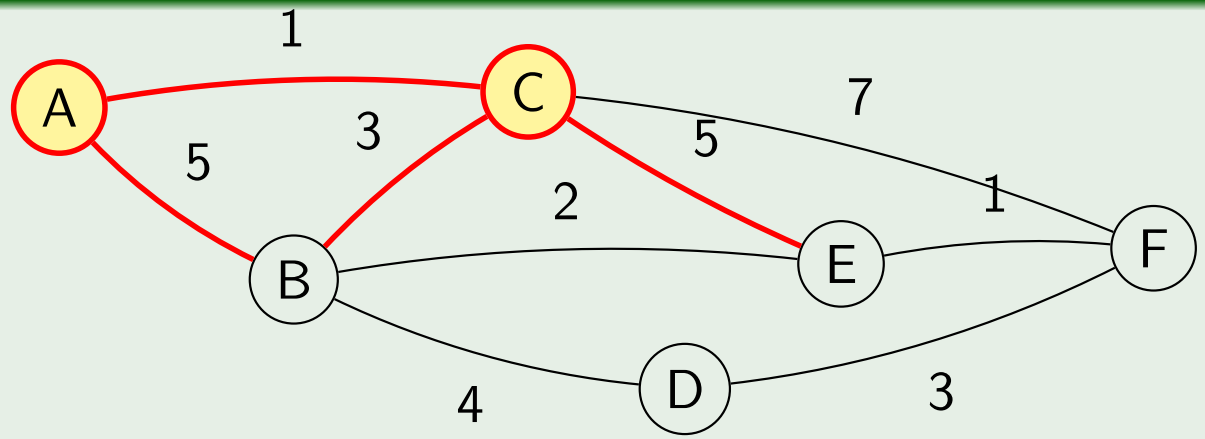
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔		1 (A)				C
✔		✔				
✔		✔				
✔		✔				
✔		✔				

Algorithme de Djikstra : exemple



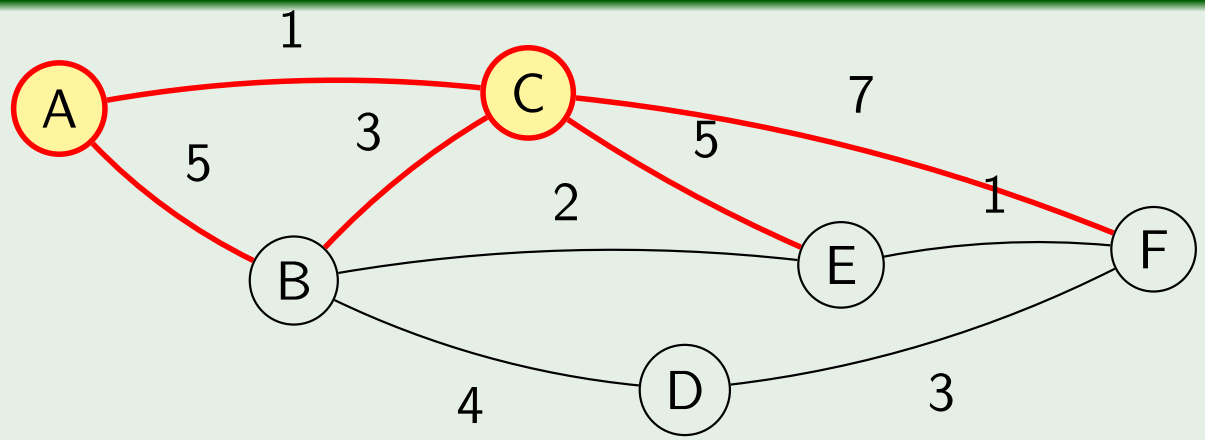
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)				C
✔		✔				
✔		✔				
✔		✔				
✔		✔				

Algorithme de Djikstra : exemple



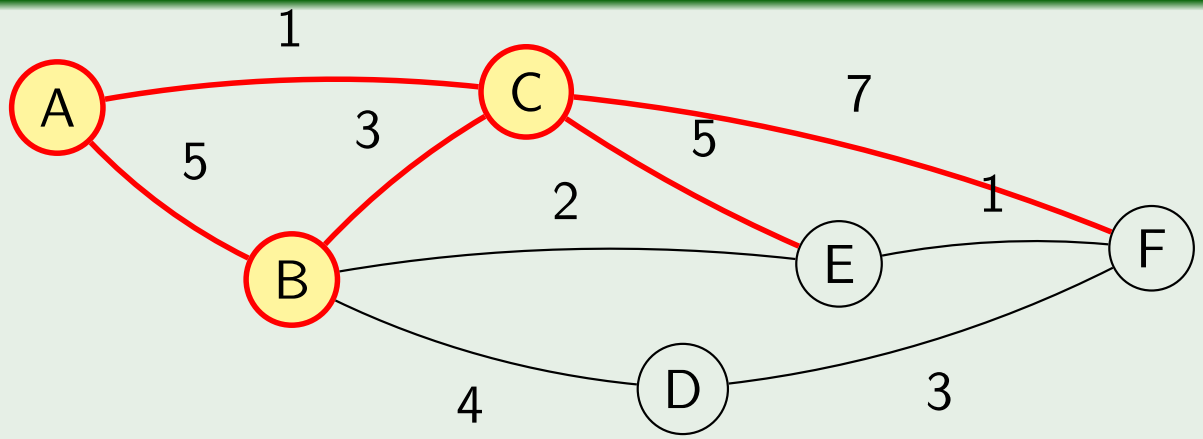
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)		C
✔		✔				
✔		✔				
✔		✔				
✔		✔				

Algorithme de Djikstra : exemple



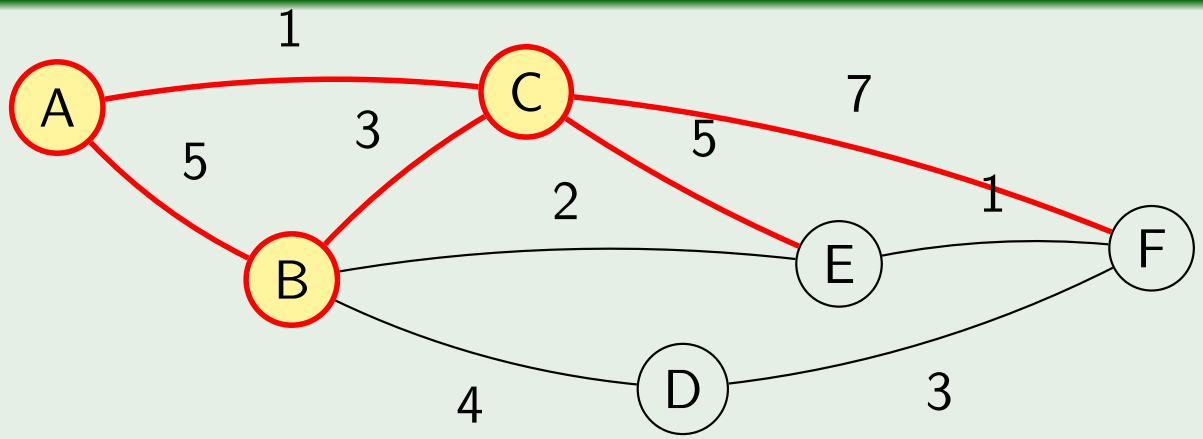
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔		✔				
✔		✔				
✔		✔				
✔		✔				

Algorithme de Djikstra : exemple



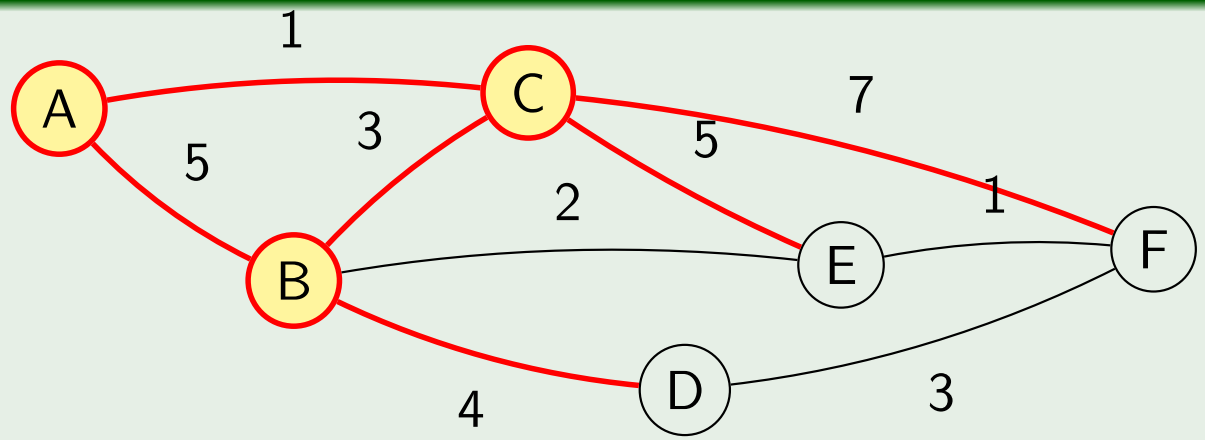
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔				
✔		✔				
✔		✔				
✔		✔				

Algorithme de Djikstra : exemple



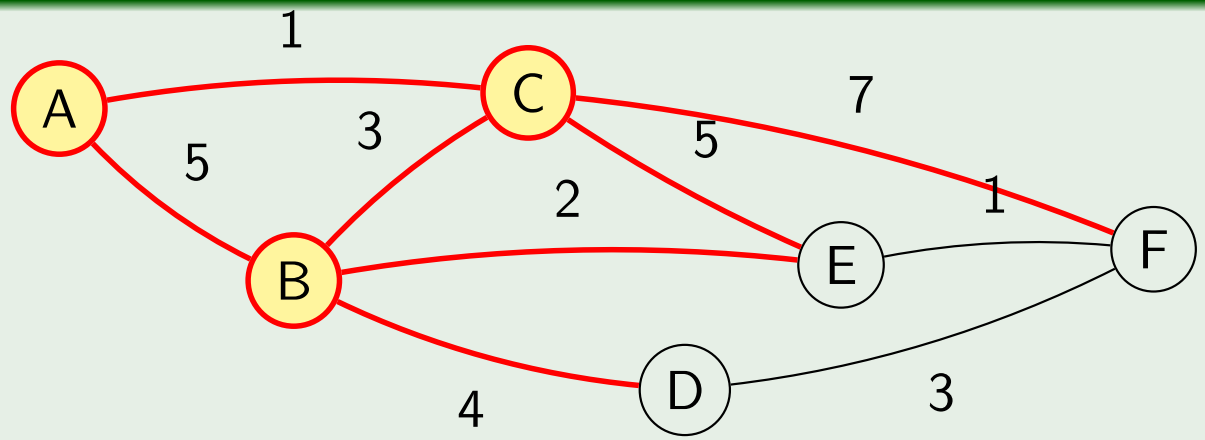
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔				B
✔	✔	✔				
✔	✔	✔				
✔	✔	✔				

Algorithme de Djikstra : exemple



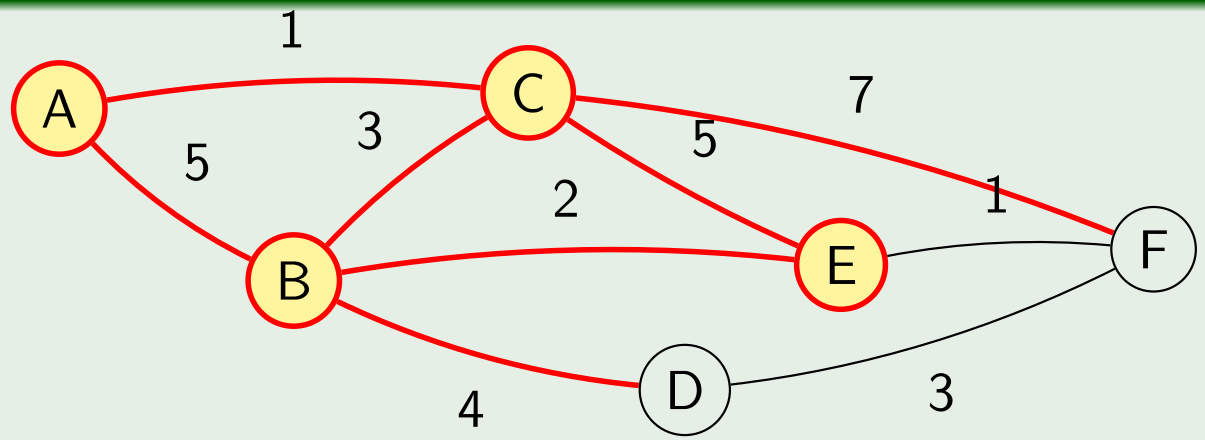
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)		8 (C)	B
✔	✔	✔				
✔	✔	✔				
✔	✔	✔				

Algorithme de Djikstra : exemple



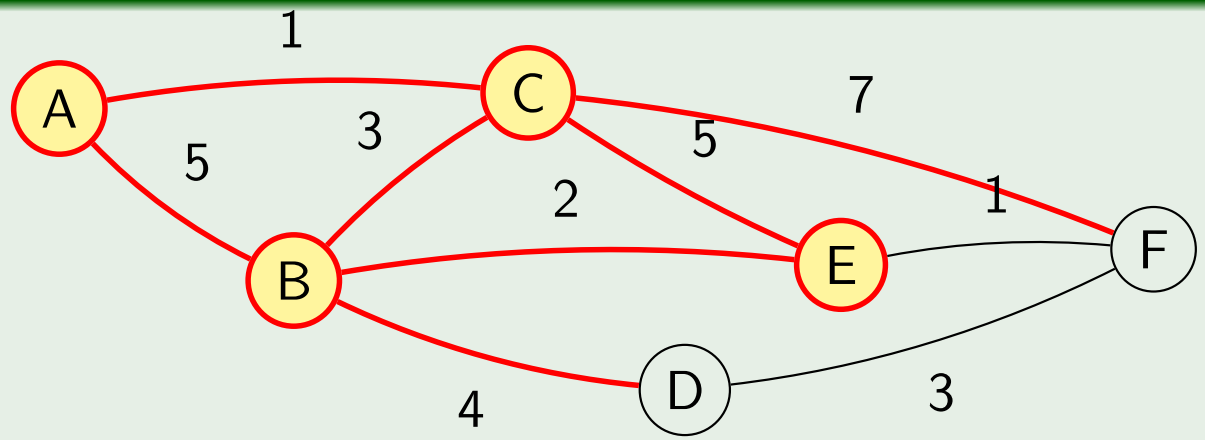
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔				
✔	✔	✔				
✔	✔	✔				

Algorithme de Djikstra : exemple



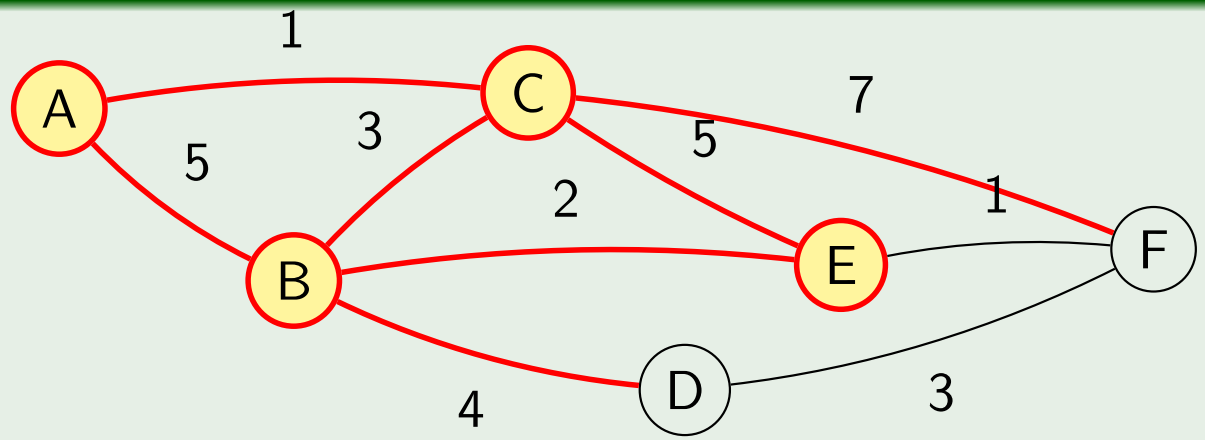
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	
✔	✔	✔				
✔	✔	✔				

Algorithme de Djikstra : exemple



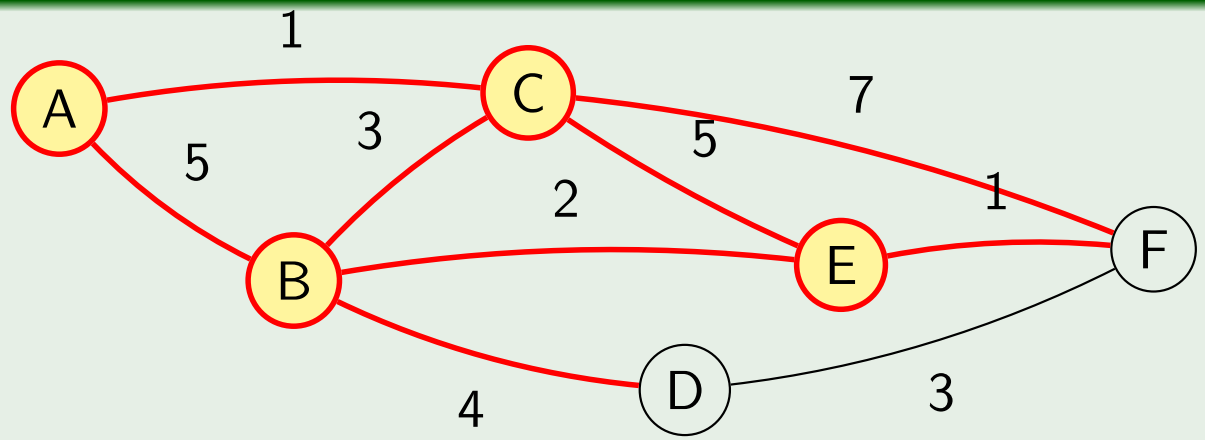
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	E
✔	✔	✔		✔		
✔	✔	✔		✔		

Algorithme de Djikstra : exemple



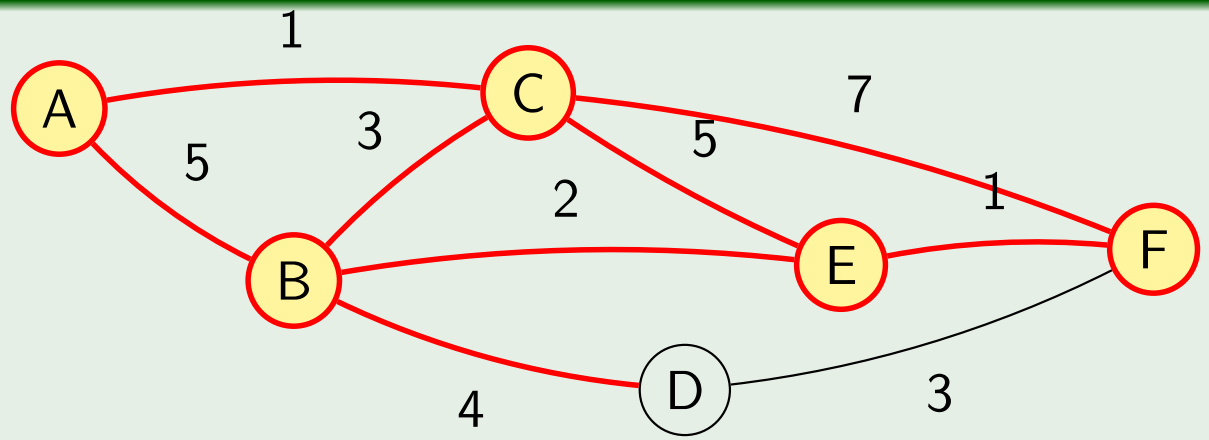
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	E
✔	✔	✔		✔		
✔	✔	✔		✔		

Algorithme de Djikstra : exemple



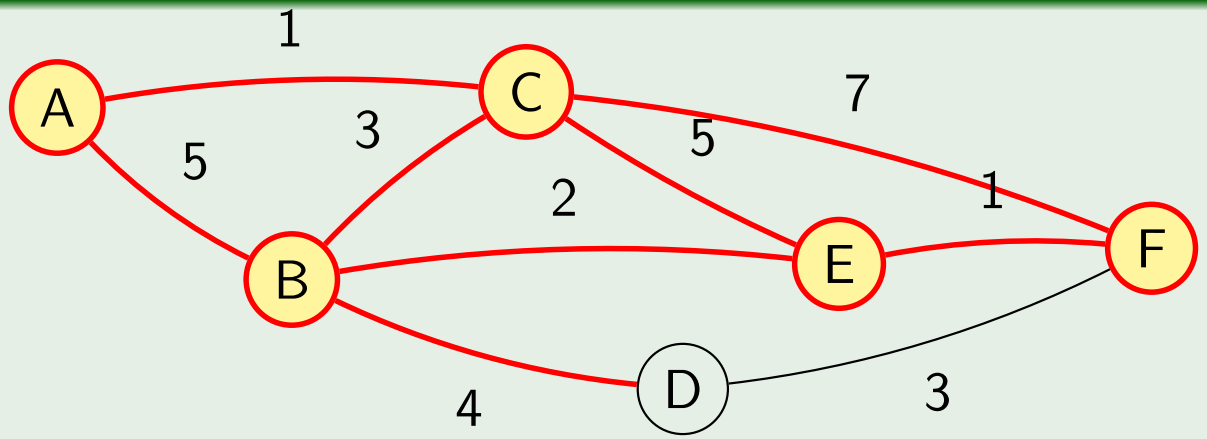
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	E
✔	✔	✔		✔		
✔	✔	✔		✔		

Algorithme de Djikstra : exemple



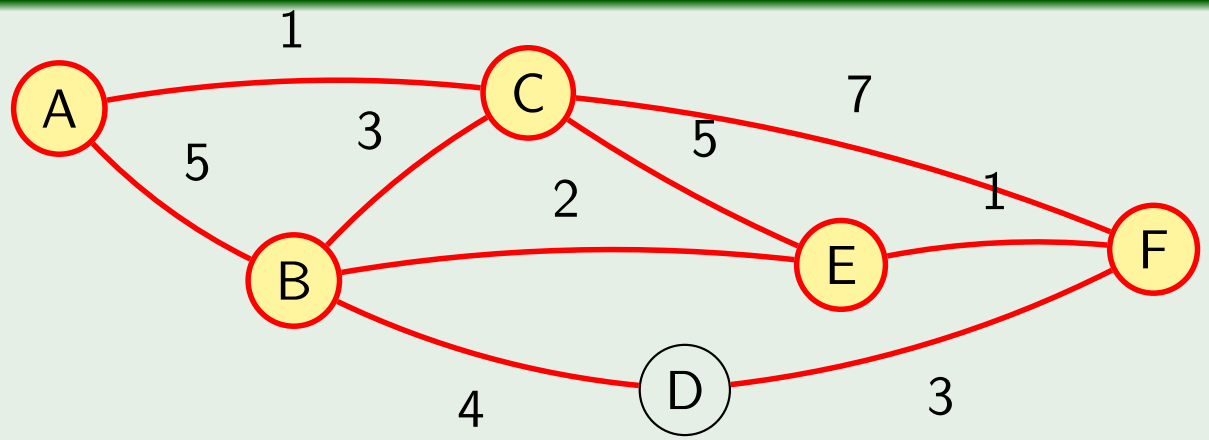
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	E
✔	✔	✔		✔	7 (E)	
✔	✔	✔		✔	✔	

Algorithme de Djikstra : exemple



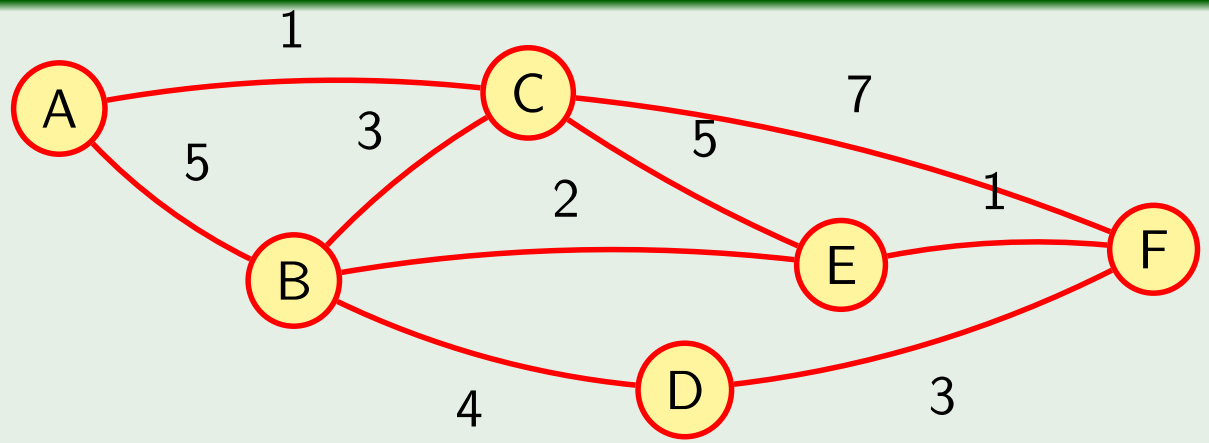
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	E
✔	✔	✔		✔	7 (E)	F
✔	✔	✔		✔	✔	

Algorithme de Djikstra : exemple



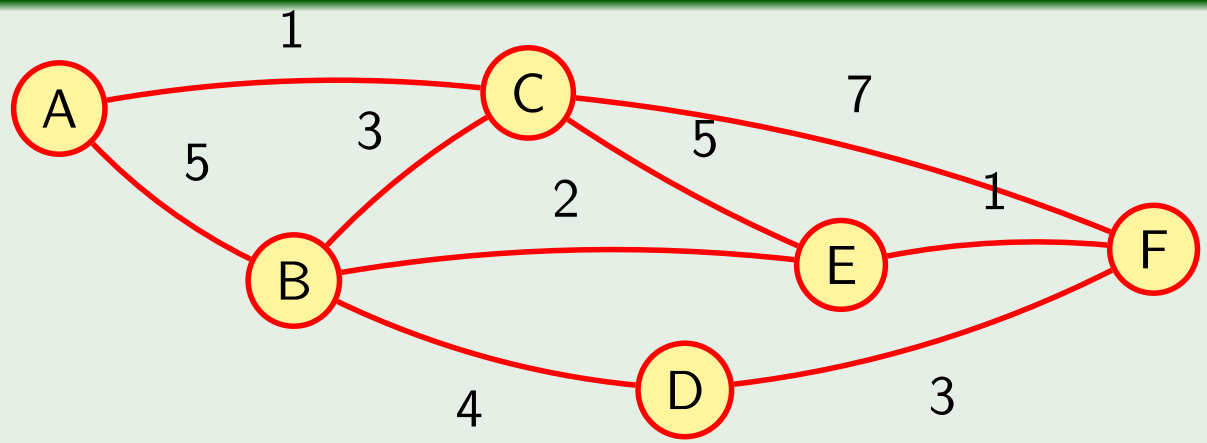
A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	E
✔	✔	✔	8 (B)	✔	7 (E)	F
✔	✔	✔		✔	✔	

Algorithme de Djikstra : exemple



A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	E
✔	✔	✔	8 (B)	✔	7 (E)	F
✔	✔	✔	8 (B)	✔	✔	

Algorithme de Djikstra : exemple



A	B	C	D	E	F	
0 (A)	5 (A)	1 (A)				A
✔	4 (C)	1 (A)		6 (C)	8 (C)	C
✔	4 (C)	✔	8 (B)	6 (B)	8 (C)	B
✔	✔	✔	8 (B)	6 (B)	7 (E)	E
✔	✔	✔	8 (B)	✔	7 (E)	F
✔	✔	✔	8 (B)	✔	✔	D

Principe

- L'algorithme de Djikstra recherche le plus court chemin depuis un sommet de départ s vers *tous les autres* sommets du graphe.

Principe

- L'algorithme de Djikstra recherche le plus court chemin depuis un sommet de départ s vers *tous les autres* sommets du graphe.
- Les poids sont supposés positifs.

Principe

- L'algorithme de Djikstra recherche le plus court chemin depuis un sommet de départ s vers *tous les autres* sommets du graphe.
- Les poids sont supposés positifs.
- C'est un adaptation du parcours en largeur, dans laquelle la file des sommets en attente de traitement est une file de priorité.

Principe

- L'algorithme de Djikstra recherche le plus court chemin depuis un sommet de départ s vers *tous les autres* sommets du graphe.
- Les poids sont supposés positifs.
- C'est un adaptation du parcours en largeur, dans laquelle la file des sommets en attente de traitement est une file de priorité.
- La priorité est la distance minimale trouvée pour le moment depuis le sommet de départ.

  Idéalement, on doit donc disposer d'une file de priorité où la mise à jour de la priorité d'un élément est disponible. Pour notre implémentation, par souci de simplicité, on adoptera une file de priorité « traditionnelle » et on enfilera plusieurs fois le même sommet (avec des priorités différentes). Et on utilisera un tableau de booléens afin de ne pas traiter deux fois le même sommet. Cela a des conséquences en terme de complexité mais non significatives à notre niveau.

Proposition d'implémentation

```
let djikstra graphe depart =
  let a_traiter = cree_file graphe.size 0 in
  let distance = Array.make graphe.size Int.max_int in
  let deja_vu = Array.make graphe.size false in
  distance.(depart) <- 0;
  enfile (0,depart) a_traiter;
  while not (est_vide a_traiter) do
    let dist, sommet_courant = defile a_traiter in
    if (not deja_vu.(sommet_courant)) then
    (deja_vu.(sommet_courant) <- true;
    List.iter (fun (w,s) -> if (distance.(sommet_courant)+w <
    ↪  distance.(s)) then
    (distance.(s) <- distance.(sommet_courant)+w; enfile
    ↪  (distance.(s),s) a_traiter); )
    (graphe.ladj.(sommet_courant)););
  done;
  distance;;
```

Complexité

- En utilisant une file de priorité « traditionnelle », chaque sommet peut être enfilé autant de fois que son degré entrant, la file contient donc au plus $|A|$ éléments.

Complexité

- En utilisant une file de priorité « traditionnelle », chaque sommet peut être enfilé autant de fois que son degré entrant, la file contient donc au plus $|A|$ éléments.
- On peut raisonnablement supposer que la complexité des opérations `enfiler` et `defiler` est logarithmique (on a vu que c'est le cas si la file est implémentée par un tas binaire). Donc la complexité des opérations sur la file ont un coût maximal $O(\log |A|)$.

Complexité

- En utilisant une file de priorité « traditionnelle », chaque sommet peut être enfilé autant de fois que son degré entrant, la file contient donc au plus $|A|$ éléments.
- On peut raisonnablement supposer que la complexité des opérations `enfiler` et `defiler` est logarithmique (on a vu que c'est le cas si la file est implémentée par un tas binaire). Donc la complexité des opérations sur la file ont un coût maximal $O(\log |A|)$.
- La complexité totale de l'algorithme est donc un $O(|A| \log |A|)$. En remarquant que $O(\log |A|) = O(\log |S|)$ (car $|A| \leq |S|^2$), on obtient une complexité $O(|A| \log |S|)$.

Complexité

- En utilisant une file de priorité « traditionnelle », chaque sommet peut être enfilé autant de fois que son degré entrant, la file contient donc au plus $|A|$ éléments.
- On peut raisonnablement supposer que la complexité des opérations `enfiler` et `defiler` est logarithmique (on a vu que c'est le cas si la file est implémentée par un tas binaire). Donc la complexité des opérations sur la file ont un coût maximal $O(\log|A|)$.
- La complexité totale de l'algorithme est donc un $O(|A|\log|A|)$. En remarquant que $O(\log|A|) = O(\log|S|)$ (car $|A| \leq |S|^2$), on obtient une complexité $O(|A|\log|S|)$.

En utilisant une file de priorité avec possibilité de modifier en temps constant la priorité d'un élément, on peut obtenir une meilleure complexité

Principe

- L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique permettant de rechercher la plus courte distance entre toutes les paires de sommets d'un graphe.

Principe

- L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique permettant de rechercher la plus courte distance entre toutes les paires de sommets d'un graphe.
- Pour un graphe pondéré $G = (S, A, \omega)$ dont les sommets sont numérotés à partir de 1, l'idée de l'algorithme est de construire successivement (pour $k = 0, \dots |S|$) les matrices $M(i, j, k)$ qui donnent en ligne i colonne j, la plus courte distance des chemins entre les sommets i et j qui n'utilisent que les sommets intermédiaires de numéros inférieurs ou égaux à k.
  Par exemple, $M(3, 5, 2)$ est la plus courte distance entre les sommets 3 et 5 qui n'utilisent que les sommets intermédiaires 1 et 2.

Exercice

Pour un graphe $G = (S, A, \omega)$ dont les sommets sont numérotés à partir de 1, on note $\delta(i, j)$ la longueur minimale d'un chemin entre les sommets i et j et on note pose $\delta(i, j) = +\infty$ si aucun chemin n'existe entre i et j.

1. Que vaut $M(i, j, |S|)$?

Exercice

Pour un graphe $G = (S, A, \omega)$ dont les sommets sont numérotés à partir de 1, on note $\delta(i, j)$ la longueur minimale d'un chemin entre les sommets i et j et on note pose $\delta(i, j) = +\infty$ si aucun chemin n'existe entre i et j.

1. Que vaut $M(i, j, |S|)$?
2. Déterminer $M(i, j, 0)$ pour tout $(i, j) \in [\![1; |S|]\!]$.

Exercice

Pour un graphe $G = (S, A, \omega)$ dont les sommets sont numérotés à partir de 1, on note $\delta(i, j)$ la longueur minimale d'un chemin entre les sommets i et j et on note pose $\delta(i, j) = +\infty$ si aucun chemin n'existe entre i et j.

1. Que vaut $M(i, j, |S|)$?
2. Déterminer $M(i, j, 0)$ pour tout $(i, j) \in [\![1; |S|]\!]$.
3. Pour tout $(i, j) \in [\![1; |S|]\!]$ et $k = 1 \dots |S|$, établir la relation de récurrence liant $M(i, j, k)$ à $M(i, j, k-1)$, $M(i, k, k-1)$ et $M(k, j, k-1)$.

Exercice

Pour un graphe $G = (S, A, \omega)$ dont les sommets sont numérotés à partir de 1, on note $\delta(i, j)$ la longueur minimale d'un chemin entre les sommets i et j et on note pose $\delta(i, j) = +\infty$ si aucun chemin n'existe entre i et j.

1. Que vaut $M(i, j, |S|)$?
2. Déterminer $M(i, j, 0)$ pour tout $(i, j) \in [\![1; |S|]\!]$.
3. Pour tout $(i, j) \in [\![1; |S|]\!]$ et $k = 1 \dots |S|$, établir la relation de récurrence liant $M(i, j, k)$ à $M(i, j, k-1)$, $M(i, k, k-1)$ et $M(k, j, k-1)$.
4. Déduire des questions précédentes une implémentation itérative (ascendante) en OCaml de l'algorithme de Floyd-Warshall.

Exercice

Pour un graphe $G = (S, A, \omega)$ dont les sommets sont numérotés à partir de 1, on note $\delta(i, j)$ la longueur minimale d'un chemin entre les sommets i et j et on note pose $\delta(i, j) = +\infty$ si aucun chemin n'existe entre i et j.

1. Que vaut $M(i, j, |S|)$?
2. Déterminer $M(i, j, 0)$ pour tout $(i, j) \in [\![1; |S|]\!]$.
3. Pour tout $(i, j) \in [\![1; |S|]\!]$ et $k = 1 \dots |S|$, établir la relation de récurrence liant $M(i, j, k)$ à $M(i, j, k-1)$, $M(i, k, k-1)$ et $M(k, j, k-1)$.
4. Déduire des questions précédentes une implémentation itérative (ascendante) en OCaml de l'algorithme de Floyd-Warshall.
5. Quelle est la complexité de l'algorithme de Floyd-Warshall ?