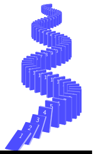
Leçon d'informatique : Recherche du plus court chemin : algorithme de Dijkstra

S. Benlhajlahsen - PCSI₁



Sommaire

I Recherche du plus court chemin 1

II Algorithme de Dijkstra 3

III Une implémentation plus simple 5

Extrait du programme :

Notions	Commentaires
Recherche d'un plus court chemin dans un graphe pondéré avec des poids positifs.	Algorithme de Dijkstra. On peut se contenter d'un modèle de file de priorité naïf pour extraire l'élément minimum d'une collection. Sur des exemples, on s'appuie sur l'algorithme A* vu comme variante de celui de Dijkstra pour une première sensibilisation à la notion d'heuristique.

I Recherche du plus court chemin

I.A Exemple de présentation

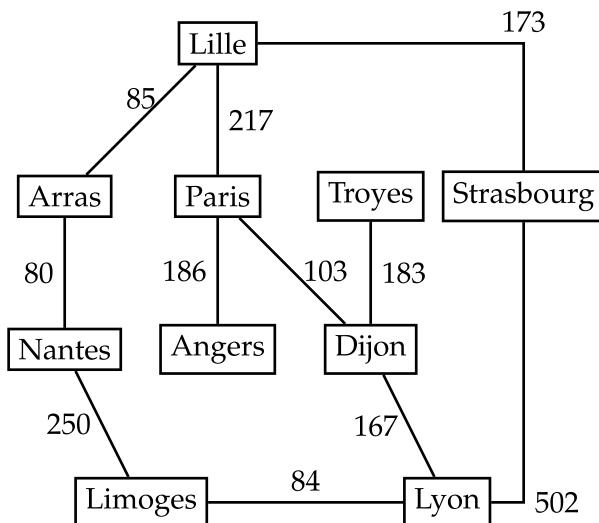


FIGURE 1 – Carte de France. Les distances n'ont pas été respectées.

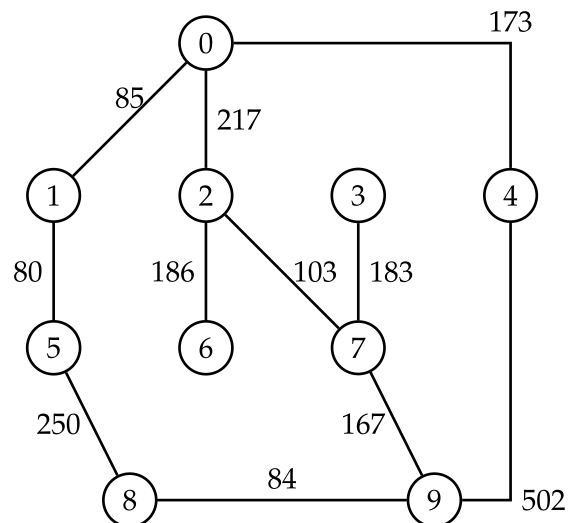


FIGURE 2 – Graphe équivalent où les sommets sont numérotés de 0 à 9.

Pour introduire le plus court chemin, on prend l'exemple du graphe de la figure 1 représentant une carte de routes de France. On cherche le plus chemin allant de Lille à Lyon. Pour nous simplifier la tâche, on prendra des sommets numérotés (voir figure 2). En tâtonnant, on se convainc rapidement que le chemin le plus rapide est : Lille → Paris → Dijon → Lyon [1] et correspond à une distance de 487 km.

Matrice et liste d'adjacence : On peut choisir de représenter ce graphe par la matrice d'adjacence ci-dessous. Comme le graphe est composé de $n = 10$ sommets, alors cette matrice appartient à $\mathcal{M}_{10}(\mathbb{R})$.
On peut aussi la représenter par la liste d'adjacence `L1` ci-dessous.

1. Ou encore 0 → 2 →7 → 9

$$M_1 = \begin{pmatrix} \infty & 85 & 217 & \infty & 173 & \infty & \infty & \infty & \infty & \infty \\ 85 & \infty & \infty & \infty & \infty & 80 & \infty & \infty & \infty & \infty \\ 217 & \infty & \infty & \infty & \infty & \infty & 186 & 103 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 183 & \infty & \infty \\ 173 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 502 \\ \infty & 80 & \infty & \infty & \infty & \infty & \infty & \infty & 250 & \infty \\ \infty & \infty & 186 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 103 & 183 & \infty & \infty & \infty & \infty & \infty & 167 \\ \infty & \infty & \infty & \infty & \infty & 250 & \infty & \infty & \infty & 84 \\ \infty & \infty & \infty & \infty & 502 & \infty & \infty & 167 & 84 & \infty \end{pmatrix}$$

```
L1=[[[1, 85], [2, 217], [4, 173]],
    [[0, 85], [5, 80]],
    [[0, 217], [6, 186], [7, 103]],
    [[7, 183]],
    [[0, 173], [9, 502]],
    [[1, 80], [8, 250]],
    [[2, 186]],
    [[2, 103], [3, 183], [9, 167]],
    [[5, 250], [9, 84]],
    [[4, 502], [7, 167], [8, 84]]]
```

I.B On essaye un algorithme glouton ?

Heuristique gloutonne : Il est bien évidemment hors de question de faire une recherche **exhaustive** de tous les chemins possibles. Ce type d'algorithme appelé `force brute` a une très mauvaise complexité.
Cela nous confronte aux même problème que celui du **sac à dos**. On peut donc tenter une heuristique gloutonne : choisir à chaque étape, le sommet le plus proche.
Dans le cas présent, cela donnerait le chemin `(0, 1, 5, 8, 9)` qui 499 km au lieu de 489.

Remarque : On rappelle qu'une heuristique est une méthode de calcul qui donne un résultat rapidement mais qui n'est pas pas nécessairement une solution optimale. La méthode gloutonne nous fait faire « le meilleur choix » à chaque étape mais ne garantit pas de donner un optimum global.

Conclusion : Une **méthode gloutonne classique ne donnera pas systématiquement le chemin le plus court.**

I.C Pile, file et file de priorité

On rappelle que :

- le parcours en profondeur (DFS) utilise une **pile** pour déterminer les sommets à visiter ;
- le parcours en largeur (BFS) utilise une **file** pour déterminer les sommets à visiter ;

Ces deux structures de données (piles et files) ne permettent pas de choisir un sommet en fonction de la distance. Il faut donc introduire une priorité dans la sortie de la liste de visite.

I.D Phase exploration

Lorsqu'on est à un sommet du graphe, on explorera prioritairement le sommet le plus proche.

Phase d'exploration et file de priorité : Partant de Paris, on peut explorer (ou visiter) les sommets 1 (Arras), 2 (Paris) ou 4 (Strasbourg). Cette liste de visite ne sera ni une pile ni une file car le premier sommet qui doit en sortir sera le plus proche de 0 (Paris).
On construit donc une `file de priorité`. Le « premier sorti » de cette file ne sera pas le « premier arrivé » comme une file classique. Cela le rôle de la fonction `sortir_min` dans la suite (voir figure 3).
C'est ici le cas avec la liste des `peres` qui relie deux à deux les sommets du graphe par un lien de « filiation ».

Dans le programme de la figure 3, on détermine le sommet le plus proche du départ parmi une liste de visite. Par exemple, la syntaxe :

`sortir_min([1,2,3], [0, 85, 217, 173])`

veut dire qu'on cherche parmi les 3 sommets `1`, `2` et `3` sachant qu'ils sont respectivement à 85, 217 et 173 km. Cette fonction doit renvoyer 1 (sommet le plus proche) et l'enlever de la liste de visite (`[1,2,3]` devient `[2,3]`).

I.E Phase d'actualisation

Remarque : À ce stade, on a un algorithme glouton. On va s'en écarter en nous laissant la possibilité de revenir sur nos pas si on a pris la « le mauvais chemin ».

Actualisation du chemin : On voit dans l'exemple proposé qu'on va commencer par choisir le sommet 1 (Arras) puis le sommet 6 (Nantes) : on aura alors parcouru 165 km. Si on choisit Limoges, on va parcourir 250 km en plus ce qui est beaucoup plus que le chemin entre Lille et Strasbourg (173).
Contrairement à un algorithme glouton pur, on se donne la possibilité de revenir sur une décision. Cela nécessitera d'actualiser le chemin. On devra donc :

```python
def sortir_min(liste_visite, distances):
    #  Attention à bien distinguer i et liste_visite[i]
    dmin = inf                    # la distance est forcément plus petite que inf
    imin = 0                      # indice du sommet le plus proche dans liste_viste
    for i in range(len(liste_visite)):
        if distances[liste_visite[i]] < dmin:
                imin = i
                dmin = distances[i]
    sommet = liste_visite[imin]                          # attention à  pop après
    liste_visite.pop(imin)               # on enleve ce sommet de la liste de visite
    return sommet, liste_visite
```

FIGURE 3

- connaître pour le « sommet actuel » le sommet le plus proche et accessible[a].
- connaitre quelle est la distance qu'on a parcouru jusqu'à présent depuis le départ jusqu'à chacun des sommets visités (liste `distances`).
- connaître les sommets déjà visités pour n'y passer qu'une seule fois (voir liste `deja_visite`).
- connaître la filiation : Lille est l'antécédent ou le père de Paris. Cette filiation peut changer car Strasbourg sera le premier père de Lyon mais mais celui-ci changera pour être Dijon à la fin (voir liste `peres`).

a. c'est-à-dire pas encore visité.

II Algorithme de Dijkstra

II.A Principe

Principe sur l'algorithme de Dijkstra : Soit n le nombre de sommets du graphe. On cherche le chemin de longueur minimale entre deux sommets `depart` et `arrivee`.

- On initialise une liste de taille n appelée `distances` qui représente la distance parcoure entre le sommet de départ et chacun des sommets libres. Cette distance est initialisée à 0 pour le sommet de départ et à `inf` pour les autres sommets.
- On construit une liste `peres` qui permet de connaître le prédécesseur de chaque sommet.
- On notera qu'on ne passe qu'une seule fois par chaque sommet. Il faut donc consigner la liste des sommets disponibles.
- Enfin, on crée une file `a_visiter` initialement composée de tous les sommets. On en enlèvera des sommets au fur et à mesure.

  Tant qu'on n'a pas encore atteint `arrivee` :
  - On appelle k_1 le sommet actuel. On commence évidemment par $k_1 =$ `depart`.
  - phase d'actualisation : Parmi les sommets encore disponibles, on vérifie si la route passant par k_1 n'est pas plus rapide qu'une éventuelle route précédemment choisie. On oublie pas d'actualiser alors la distance parcourue jusque k_1 et de consigner que k_1 est le nouveau père des sommets disponibles et atteignables.
  - phase d'exploration : On fait à présent sortir (*pop*) le sommet k_2 correspondant à la plus faible distance parcourue depuis le sommet de départ.
    - si k_1 et k_2 sont identiques, alors l'arrivée n'est pas atteignable.
    - sinon, on rend k_2 indisponible pour la suite et k_1 prend la valeur de k_2.
- phase de remontée : on remonte la filiation établie précédemment.

II.B Faisons à la main

On peut d'abord appliquer cet algorithme à la main entre le sommet 0 et le sommet 9.

Itération	k_1	k_2	peres	distances
0	0 (Lille)		[, , , , , , , , ,]	$[0, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty]$
1	0 (Lille)	1 (Arras)	[, 0, 0, , 0, , , , ,]	$[0, 85, 217, \infty, 173, \infty, \infty, \infty, \infty, \infty]$
2	1 (Arras)	2 (Paris)	[, 0, 0, , 0, 1, , , ,]	$[0, 85, 217, \infty, 173, 165, \infty, \infty, \infty, \infty]$
3	2 (Paris)	4 (Strasbourg)	[, 0, 0, , 0, 1, 2, 2, ,]	$[0, 85, 217, \infty, 173, 165, 403, 320, \infty, \infty]$
4	4 (Strasbourg)	5 (Nantes)	[, 0, 0, , 0, 1, 2, 2, , 4]	$[0, 85, 217, \infty, 173, 165, 403, 320, \infty, 675]$
5	5 (Nantes)	6 (Angers)	[, 0, 0, , 0, 1, 2, 2, 5, 4]	$[0, 85, 217, \infty, 173, 165, 403, 320, 415, 675]$
6	6 (Angers)	7 (Dijon)	[, 0, 0, , 0, 1, 2, 2, 5, 4]	$[0, 85, 217, \infty, 173, 165, 403, 320, 415, 675]$
7	7 (Dijon)	3 (Troyes)	[, 0, 0, 7, 0, 1, 2, 2, 5, 7]	$[0, 85, 217, 503, 173, 165, 403, 320, 415, 487]$
8	3 (Troyes)	8 (Limoges)	[, 0, 0, 7, 0, 1, 2, 2, 5, 7]	$[0, 85, 217, 503, 173, 165, 403, 320, 415, 487]$
9	8 (Limoges)	9 (Lyon)	[, 0, 0, 7, 0, 1, 2, 2, 5, 7]	$[0, 85, 217, 503, 173, 165, 403, 320, 415, 487]$

Commentaires :

- Partant de Lille, on peut directement accéder à Arras, Paris et Strasbourg. Lille sera donc leur antécédent (itération 1) ;
- Le sommet le plus proche est Arras que l'on explore (actualisation de k_2).
- Depuis Arras, on peut aller à Nantes. Cependant, si on va à Nantes, on parcourra 165 km ce qui est supérieur au chemin entre Lille et Paris ou entre Lille et Strasbourg.
- On « rebrousse » chemin et on essaye d'explorer Paris ou Strasbourg.
- On notera qu'à la7ème itération, on remplace le chemin passant par Strasbourg (4) et on met le chemin passant par Dijon (7). On actualise alors les listes `peres` et `distances`.

Exercice 1 : Refaire le même travail pour trouver le plus court chemin entre Troyes (3) et Nantes (5)

Itération	k_1	k_2	peres	distances
0	3 (Troyes)		[, , , , , , , , , ,]	[, , , , , , , , , ,]
1			[, , , , , , , , , ,]	[, , , , , , , , , ,]
2			[, , , , , , , , , ,]	[, , , , , , , , , ,]
3			[, , , , , , , , , ,]	[, , , , , , , , , ,]
4			[, , , , , , , , , ,]	[, , , , , , , , , ,]
5			[, , , , , , , , , ,]	[, , , , , , , , , ,]
6			[, , , , , , , , , ,]	[, , , , , , , , , ,]
7			[, , , , , , , , , ,]	[, , , , , , , , , ,]
8			[, , , , , , , , , ,]	[, , , , , , , , , ,]
9			[, , , , , , , , , ,]	[, , , , , , , , , ,]

Réponse : Voir figure 6 en page **??**.

II.C Implémentation

Implémentation : L'algorithme de la figure 4 propose une implémentation qui prend en argument une matrice d'adjacence et deux sommets (entiers). Elle renvoie le parcours ainsi que la distance minimale entre les deux sommets.
Si les deux sommets ne sont pas joignables, elle renvoie une distance infinie.

Remarque : On pourra vérifier que cette implémentation fonctionne aussi avec un graphe orienté.

II.D Remarques

complexité : Si on note n le nombre de sommets et p le nombre d'arêtes, l'algorithme de Dijkstra a une complexité en $o\left(a + n\ln(n)\right)$. Cela veut dire qu'on parcourt l'ensemble des arêtes (ou arcs) du graphe.
On rappelle que le parcours de graphe générique (dfs ou bfs) a un complexité en $O(n+p)$.
Ainsi, Dijkstra est un peu plus coûteux que le parcours classique.

```
def dijkstra(M, depart, arrivee):
    # Paramètres
    n = len(M)                                                            # nombre de sommets
    peres = [None for i in range(n)]                         # listes des peres dans le parcours
    distances = [inf for i in range(n)]                     # liste des distances entre la ville
                                                                          # de départ et le sommet i
    deja_visite = [False for i in range(n)]                # on cochera les sommets deja visités
                                                       # on initialise ces listes pour le départ choisi
    distances[depart] = 0                                   # La distance de départ à lui-même vaut 0
    deja_visite[depart] = True

    # k1 sera le sommet actuellement visité
    k1 = depart
    a_visiter = [i for i in range(n) if i != depart]
    while k1 != arrivee:
        # ------------------phase d'actualisation------------------------------
        for i in range(n):
            if deja_visite[i] == False:
                d = distances[k1] + M[k1][i]
                if d < distances[i]:
                    distances[i] = d
                    peres[i] = k1


        # ------------------phase d'exploration -------------------------------
        k2, a_visiter = sortir_min(a_visiter, distances)
        if k1 == k2:
            return [], inf
        else:
            deja_visite[k2] = True # on sélectionne ce sommet
            k1 = k2
    return distances, peres
```

FIGURE 4

III Une implémentation plus simple

L'implémentation précédente est peu claire. On va simplifier le problème en le découpant en sous-problème.

III.A Idée générale

Algorithme en pseudo-code :

Mettre le sommet de départ d dans la file de priorité
Tant que la file de priorité n'est pas vide :
    Prendre un sommet s de la file
    Si s n'a pas déjà été visité :
        Marquer s
        Pour tout les sommets v voisins de s :
            Si le trajet $d \to v$ est tendu :
                Relaxer/Actualiser le trajet $d \to v$ en passant par s
                Mettre v dans la file

III.B files de priorité

Á chaque instant la file de priorité va contenir une liste de sommet ainsi que la distance parcourue depuis le départ jusqu'à ce ce sommet. Ainsi, dans l'exemple du trajet de Lille (0) vers Lyon (9) :

- la file de priorité sera au début : `[(0,0)]`. On part de Lille en ayant parcouru 0 km.
- à l'itération suivante, on *pop* Lille et on ajoute Arras, Paris et Strasbourg. Cela donne une file de priorité :

`[(2, 217), (4, 173), (1, 85)]`

> **Remarque :** C'est une liste composée de `tuple`. On notera que la file est triée par ordre décroissante des distances parcourues depuis le départ.

Dans le code ci-dessous :

```python
def nouvelle_file_prio():
    return []
def file_prio_vide(L):
    return len(L)==0
def file_prio_pop(L):
    return L.pop()

def position_insertion (liste, element):
    for i in range(len(liste)):
        if liste[i][1]<element[1]:
            return i
    return len(liste)
def file_prio_push(L, element):
    position = position_insertion(L,element)
    L.insert(position,element)
```

- on peut créer une file vide ;
- vérifier que la liste est vide ;
- faire *pop* sur le dernier élément et le renvoyer. Comme la file est triée de la gauche à droite, le dernier élément est prioritaire.
- déterminer à quelle position insérer un élément en fonction de la valeur de la distance parcourue d'où `element[1]`
- faire `push` à l'aide de `insert`. préalablement

III.C arête tendue et actualisation

On va simplifier la phase d'actualisation. Prenons l'exemple de la figure 5. Après deux étapes du parcours, on découvre les sommets 2 et 3, voisins du sommet 1, donc :

Sommet	0	1	2	3
Distance	0	30	80	50
Père	`None`	0	1	1

Donc le plus court chemin trouvé à ce stade de 0 vers 2 est le chemin $0 \to 1 \to 2$ et est de longueur totale 8. Mais au tour de boucle suivant on considère le sommet 3, et on découvre que le chemin $0 \to 1 \to 3 \to 2$ est de longueur 6, donc plus court que celui trouvé précédemment :

Sommet	0	1	2	3
Distance	0	30	**60**	50
Père	`None`	0	**3**	1

On dira que l'arête $1 \to 2$ est tendu et qu'elle doit être relaxée/actualisée.

```python
def est_tendu(u, v, poids, distances):
    return distances[u]+poids < distances[v]
def relaxation(u, v, poids, distances, peres):
    distances[v] = distances[u]+poids
    peres[v] = u
```

On écrit donc :

- une première fonction `est_tendu` qui renvoie `True` ou `False`. Pour deux sommets `u` et `v` et connaissant le poids/-distance entre ces deux sommets, on cherche à savoir si le nouveau chemin menant à `v` et passant par `u` est plus court que l'ancien chemin menant à `v`.
- En fonction du résultat précédant, on actualise en modifiant les listes `peres` et `distances`.

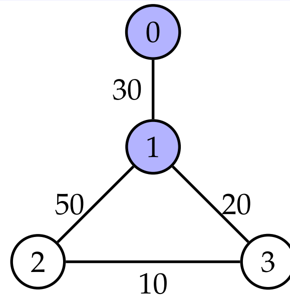


FIGURE 5

III.D Implémentation

On donne ci-dessous une implémentation de dijkstra avec une liste d'adjacence :

```
def init_dijkstra(graphe):
    n = len(graphe)
    distances = [inf for _ in range(n)]
    peres = [None for _ in range(n)]
    deja_visite = [False for _ in range(n)]
    file_prio = nouvelle_file_prio()
    return distances, peres, deja_visite, file_prio

def dijkstra(graphe, depart):
    distances, peres, deja_visite, fp =  init_dijkstra(graphe)
    distances[depart] = 0
    file_prio_push(fp, (depart, distances[depart]))
    while file_prio_vide(fp)==False:
        sommet = file_prio_pop(fp)[0]
        if deja_visite[sommet]==False:
            deja_visite[sommet] = True
            for x in graphe[sommet]:
                voisin = x[0]
                poids = x[1]
                if est_tendu(sommet, voisin, poids, distances):
                    relaxation(sommet, voisin, poids, distances, peres)
                file_prio_push(fp, (voisin, distances[voisin])  )
    return distances, peres
```

Commentaire :

- On commence par initialiser toutes les listes.
- On marque le depart du parcours
- A chaque itération, on fait sortir le sommet prioritaire (c'est-à-dire celui correspondant à distance parcourue la plus faible depuis le départ) :
  - s'il n'a pas déjà été visité, on le marque.
  - On explore les voisins de ce sommet ;
  - On actualise les distances si passer par ce sommet est plus court pour aller au voisin.

III.E Quelques applications

Distance parcourue entre 2 sommets : Une fois écrite l'implémentation précédente. On note que celle-ci renvoie les distances les plus courtes depuis le départ. On trouve donc facilement distance entre ce départ et n'importe quel sommet.

```
def distance_parcourue(graphe, depart, arrivee):
    distances, peres = dijkstra(graphe, depart)
    return distances[arrivee]
```

Itinéraire le plus court entre deux sommets : Comme on a la liste `peres`, on peut déterminer l'itinéraire le plus court depuis le départ en remontant « la filiation ».

```python
def chemin(graphe, depart, arrivee):
    distances, peres = dijkstra(graphe, depart)
    if peres[arrivee]==None:
        return []
    else:
        itineraire = [arrivee]
        sommet = arrivee
        while sommet!=depart:
            sommet = peres[sommet]
            itineraire.append(sommet)
        return itineraire[::-1]    #On inverse la liste
```

Itération	k_1	k_2	`peres`	`distances`
k1=	3 (Troyes)	7 (Dijon)	[, , , , , , , 3, ,]	[∞, ∞, ∞, 0, ∞, ∞, ∞, 183, ∞, ∞]
k1=	7 (Dijon)	2 (Paris)	[, , 7, , , , , 3, , 7]	[∞, ∞, 286, 0, ∞, ∞, ∞, 183, ∞, 350]
k1=	2 (Paris)	9 (Lyon)	[2, , 7, , , , 2, 3, , 7]	[503, ∞, 286, 0, ∞, ∞, 472, 183, ∞, 350]
k1=	9 (Lyon)	8 (Limoges)	[2, , 7, , 9, , 2, 3, 9, 7]	[503, ∞, 286, 0, 852, ∞, 472, 183, 434, 350]
k1=	8 (Limoges)	6 (Angers)	[2, , 7, , 9, 8, 2, 3, 9, 7]	[503, ∞, 286, 0, 852, 684, 472, 183, 434, 350]
k1=	6 (Angers)	0 (Lille)	[2, , 7, , 9, 8, 2, 3, 9, 7]	[503, ∞, 286, 0, 852, 684, 472, 183, 434, 350]
k1=	0 (Lille)	1 (Arras)	[2, 0, 7, , 0, 8, 2, 3, 9, 7]	[503, 588, 286, 0, 676, 684, 472, 183, 434, 350]
k1=	1 (Arras)	4 (Strasbourg)	[2, 0, 7, , 0, 1, 2, 3, 9, 7]	[503, 588, 286, 0, 676, 668, 472, 183, 434, 350]
k1=	4 (Strasbourg)	5 (Nantes)	[2, 0, 7, , 0, 1, 2, 3, 9, 7]	[503, 588, 286, 0, 676, 668, 472, 183, 434, 350]

FIGURE 6