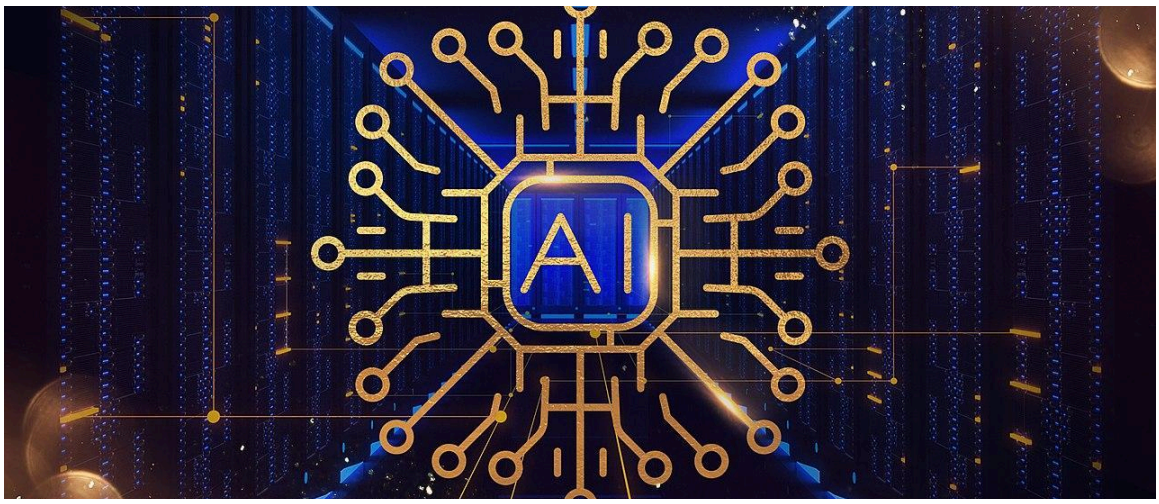

Rapport

Intelligence Artificielle

Réalisé par :

OUALID EL-OUARDI
20000591

Encadré par :

Mr. Ait lahcen Ayoub

Exploration en Largeur d'abord :

- **Résumé : Explore le graphe en visitant tous les voisins d'un nœud avant de passer aux nœuds suivants.**
- **Avantages : Garantit de trouver la solution la plus courte pour les graphes non pondérés.**
- **Inconvénients : Peut consommer beaucoup de mémoire pour les graphes très larges.**
- **Meilleure solution : Adaptée aux graphes non pondérés où la recherche de la solution la plus courte est primordiale.**

```c
static int E_largeur(GrapheMat *graphe, Liste *li, int numSommet, char but[])
{
    int nMax = graphe->nMax;
    NomSom *extraite = NULL;
    insererEnFinDeListe(li, graphe->nomS[numSommet]);
    graphe->marque[numSommet] = true;
    while (!listeVide(li))
    {
        extraite = (NomSom *)extraireEnTeteDeListe(li);
        noeudsVisite++;
        if (strcmp(*extraite, but) == 0)
        {
            printf("  ->  %s (Noeud but) \n", but);
            return 1;
        }
        numSommet = rang(graphe, *extraite);
        printf("->%s", *extraite);

        for (int i = 0; i < graphe->nMax; i++)
        {
            if ((graphe->element[numSommet * nMax + i] == vrai) && !graphe->marque[i])
            {
                insererEnFinDeListe(li, graphe->nomS[i]);
                graphe->marque[i] = vrai;
            }
        }
    }
    if (numSommet == graphe->n)
    {
        printf("\nNoeud Introuvable");
        return 0;
    }
    return 0;
}
```

graphemat.cpp > parcoursLargeurDabord(GrapheMat *)

```
253  void parcoursLargeurDabord(GrapheMat *graphe)
254  {
255      printf("\n\t\tLe Parcours en Largeur : \n\n");
256      initMarque(graphe);
257      Liste *li = creerListe();
258      char str[20];
259
260      printf("Entrez le sommet but recherche:  ");
261      scanf("%s", str);
262      printf("\n\t\tLe Chemin parcours durant notre recherche du noeud but :   \n\n");
263      int trouve = 0;
264      for (int i = 0; i < graphe->n; i++)
265      {
266          if (!graphe->marque[i])
267          {
268              trouve = E_largeur(graphe, li, i, str);
269          }
270          if (trouve == 1)
271              break;
272      }
273      printf("\n Le nombre de Noeuds visites : %d", noeudsVisite);
274      noeudsVisite = 0;
275  }
```

```
S0 S1 S2 S3 S4 S5 S6 S7 ;
S0 : S1 () S6 () ;
S1 : S2 () S3 () S5 () ;
S2 : S3 () ;
S3 : S1 () ;
S4 : S3 () S5 () S7 () ;
S5 : S1 () S3 () ;
S6 : S5 () ;
```

```
EXPLORATION DES GRAPHES

0 - Fin du programme
1 - Creation a partir d'un fichier

2 - Initialisation d'un graphe vide
3 - Ajout d'un sommet
4 - Ajout d'un arc

5 - Liste des sommets et des arcs
6 - Destruction du graphe
7 - Parcours en profondeur d'un graphe

8   - Exploration en Largeur d'abord avec but
9  - Exploration en  profondeur avec but
10  - Exploration en  profondeur Iteratif
11  - Exploration a cout uniforme
12  - Exploration A*
13  - Exploration par Escalade
14  - Exploration par recuit simule
Votre choix ? 8


                Le Parcours en Largeur :

Entrez le sommet but recherche:  S5

                Le Chemin parcours durant notre recherche du noeud but :

->S0->S1->S6->S2->S3  ->  S5 (Noeud but)

 Le nombre de Noeuds visites : 6

Taper Return pour continuer
```

Exploration en profondeur d'abord:

- **Résumé : Explore le graphe en se déplaçant aussi loin que possible le long d'une branche avant de revenir en arrière.**
- **Avantages : Utilise moins de mémoire que l'exploration en largeur d'abord.**
- **Inconvénients : Peut boucler indéfiniment dans des graphes avec des boucles, ne garantit pas de trouver la solution la plus courte.**
- **Meilleure solution : Convient pour les graphes profonds avec une grande profondeur maximale.**

graphemat.cpp > parcoursLargeurDabord(GrapheMat *)

```
278  static void profondeurDabord(GrapheMat *graphe, int numSommet, char but[])
279  {
280      int nMax = graphe->nMax;
281
282      graphe->marque[numSommet] = vrai;
283      if (strcmp(graphe->nomS[numSommet], but) == 0)
284      {
285          check = vrai;
286          printf("%s -> BUT ", but);
287          return;
288      }
289      printf("%s\n", graphe->nomS[numSommet]);
290
291      for (int i = 0; i < graphe->n; i++)
292      {
293          if ((graphe->element[numSommet * nMax + i] == vrai) && !graphe->marque[i])
294          {
295              if (check != vrai)
296                  profondeurDabord(graphe, i, but);
297          }
298      }
299  }
```

```c
void parcoursProfondDAbord(GrapheMat *graphe)
{
    initMarque(graphe);
    char str[20];

    printf("Veuiller Entrez le sommet but recherche :  ");
    scanf("%s", str);
    printf("\n\t\tLe Chemin parcours durant notre recherche du noeud but: \n");

    for (int i = 0; i < graphe->n; i++)
    {
        if (!graphe->marque[i])
        {
            profondeurDabord(graphe, i, str);
            if (check == vrai)
            {
                break;
            }
        }
    }
    if (check != vrai)
    {
        printf("\n Noeud Introuvable");
    }

    check = faux;
}
```

```
S0 S1 S2 S3 S4 S5 S6 S7 ;
S0 : S1 () S6 () ;
S1 : S2 () S3 () S5 () ;
S2 : S3 () ;
S3 : S1 () ;
S4 : S3 () S5 () S7 () ;
S5 : S1 () S3 () ;
S6 : S5 () ;
```

```
8   - Exploration en Largeur d'abord
9  - Exploration en  profondeur d'abord
10  - Exploration en  profondeur Iteratif
11  - Exploration a cout uniforme
12  - Exploration A*
13  - Exploration par Escalade
14  - Exploration par recuit simule
Votre choix ? 9

Veuiller Entrez le sommet but recherche :  S5

                Le Chemin parcours durant notre recherche du noeud but:
S0
S1
S2
S3
S5 -> BUT

Taper Return pour continuer
```

```
 8   - Exploration en Largeur d'abord
 9  - Exploration en  profondeur d'abord
 10  - Exploration en  profondeur Iteratif
 11  - Exploration a cout uniforme
 12  - Exploration A*
 13  - Exploration par Escalade
 14  - Exploration par recuit simule
 Votre choix ? 9

 Veuiller Entrez le sommet but recherche :  S19

                 Le Chemin parcours durant notre recherche du noeud but:
 S0
 S1
 S2
 S3
 S5
 S6
 S4
 S7

  Noeud Introuvable
```

Exploration itérative en profondeur:

- **Résumé : Variante de l'exploration en profondeur d'abord où la profondeur maximale est augmentée progressivement.**
- **Avantages : Combine les avantages de l'exploration en profondeur d'abord et de l'exploration en largeur d'abord.**
- **Inconvénients : Peut être inefficace pour des graphes très larges ou profonds.**
- **Meilleure solution : Appropriée pour les graphes où la profondeur maximale est inconnue et pour éviter une utilisation excessive de la mémoire.**

graphemat.cpp > profondeurLimite(GrapheMat *, int, char [], int)

```
399  static void profondeurIteratif(GrapheMat *graphe, int numSommet, int numNiveau, char but[])
400  {
401      if (!trouveS)
402      {
403          noeudsVisite++;
404          if (strcmp(graphe->nomS[numSommet], but) == 0)
405          {
406              printf("%s -> But ", but);
407              trouveS = true;
408          }
409          else if (numNiveau == 0)
410          {
411              printf("-> %s -> ", graphe->nomS[numSommet]);
412              graphe->marque[numSommet] = vrai;
413          }
414          else
415          {
416              if (numNiveau > 0)
417              {
418                  int nMax = graphe->nMax;
419                  graphe->marque[numSommet] = vrai;
420                  printf("-> %s ", graphe->nomS[numSommet]);
421                  for (int i = 0; i < graphe->n; i++)
422                  {
423                      if ((graphe->element[numSommet * nMax + i] == vrai) && !graphe->marque[i])
424                      {
425                          profondeurIteratif(graphe, i, numNiveau - 1, but);
426                      }
427                  }
428              }
429          }
430      }
431  }
```

graphemat.cpp > profondeurLimite(GrapheMat *, int, char [], int)

```
433  void parcoursProfondeurIteratif(GrapheMat *graphe)
434  {
435      char str[20];
436      int niveau, numNiveau = 0;
437      printf("Veuiller Entrez le sommet but recherche :  ");
438      scanf("%s", str);
439      printf("Entrer le niveau limite de recherche : ");
440      scanf("%d", &niveau);
441      initMarque(graphe);
442      printf("\n\t\tLe Chemin parcours durant notre recherche du noeud but:  \n\n");
443      for (numNiveau = 0; numNiveau <= niveau; numNiveau++)
444      {
445          if (trouveS)
446              break;
447          printf("Limite %d : ", numNiveau);
448          initMarque(graphe);
449          for (int i = 0; i < graphe->n; i++)
450          {
451              if (!graphe->marque[i])
452                  profondeurIteratif(graphe, i, numNiveau, str);
453              break;
454          }
455          printf("\n");
456      }
457      if (!trouveS)
458      {
459          trouveS = false;
460          printf("\nLe noeud %s est INTROUVABLE", str);
461      }
462      else
463      {
464          printf("\n\t\tLe noeud est trouve dans la limite \" %d \" *****\n\n", numNiveau - 1);
465      }
466      printf("\n Les nombre de Noeuds visites : %d", noeudsVisite);
467      noeudsVisite = 0;
468      trouveS = false;
469  }
```

```
11  - Exploration en  profondeur Iteratif
12  - Exploration a cout uniforme
13  - Exploration A*
14  - Exploration par Escalade
15  - Exploration par recuit simule
Votre choix ? 11

Veuiller Entrez le sommet but recherche :  S5
Entrer le niveau limite de recherche : 4

               Le Chemin parcours durant notre recherche du noeud but:

Limite 0 : -> S0 ->
Limite 1 : -> S0 -> S1 -> -> S6 ->
Limite 2 : -> S0 -> S1 -> S2 -> -> S3 -> S5 -> But

               Le noeud est trouve dans la limite " 2 " *****


 Les nombre de Noeuds visites : 9

Taper Return pour continuer
```

```
Votre choix ? 11

Veuiller Entrez le sommet but recherche :  S4
Entrer le niveau limite de recherche : 4

               Le Chemin parcours durant notre recherche du noeud but:

Limite 0 : -> S0 ->
Limite 1 : -> S0 -> S1 -> -> S6 ->
Limite 2 : -> S0 -> S1 -> S2 -> -> S3 -> -> S5 -> -> S6
Limite 3 : -> S0 -> S1 -> S2 -> S3 -> -> S5 -> S6
Limite 4 : -> S0 -> S1 -> S2 -> S3 -> S5 -> S6

Le noeud S4 est INTROUVABLE
 Les nombre de Noeuds visites : 22

Taper Return pour continuer
```

Exploration à coût uniforme :

- **Résumé : Explore le graphe en sélectionnant les nœuds avec le coût total le plus bas.**
- **Avantages : Garantit de trouver la solution avec le coût total le plus bas.**
- **Inconvénients : Peut être inefficace si les coûts de chemin sont très disparates.**
- **Meilleure solution : Appropriée pour les graphes avec des coûts de chemin uniformes ou légèrement différenciés.**

graphemat.cpp > profondeurLimite(GrapheMat *, int, char [], int)

```
471  static void coutUniforme(GrapheMat *graphe, Liste *li, int numSommet, int but)
472  {
473      for (int j = 0; j < 100; j++)
474      {
475          strcpy(*(chemin + j), "");
476      }
477      strcpy(*(chemin + numSommet), graphe->nomS[numSommet]);
478      int nMax = graphe->nMax;
479      Element *extraite = NULL;
480      insererEnFinDeListe(li, graphe->nomS[numSommet], cout + numSommet);
481      graphe->marque[numSommet] = vrai;
482      while (!listeVide(li))
483      {
484          extraite = (Element *)extraireEnTeteDeListe(li);
485          noeudsVisite++;
486          numSommet = rang(graphe, (char *)extraite);
487          if (numSommet == but)
488          {
489              trouveS = vrai;
490              return;
491          }
492          for (int i = 0; i < graphe->n; i++)
493          {
494              if ((graphe->element[numSommet * nMax + i] == vrai) && !graphe->marque[i])
495              {
496                  strcat(*(chemin + i), *(chemin + numSommet));
497                  strcat(*(chemin + i), "->");
498                  strcat(*(chemin + i), graphe->nomS[i]);
499                  *(cout + i) = graphe->valeur[numSommet * nMax + i] + *(cout + numSommet);
500
501                  insererEnOrdre(li, graphe->nomS[i], cout + i);
502                  graphe->marque[i] = vrai;
503              }
504              else
505              {
506                  booleen trouvee = chercherUnObjetBis(li, graphe->nomS[i]);
507                  if ((graphe->element[numSommet * nMax + i] == vrai) && trouvee && *(cout + i) > graphe->valeur[numSommet * nMax + i] + *(cout + numSommet))
508                  {
509                      *(cout + i) = graphe->valeur[numSommet * nMax + i] + *(cout + numSommet);
510                      strcpy(*(chemin + i), "");
511                      strcat(*(chemin + i), *(chemin + numSommet));
512                      strcat(*(chemin + i), "->");
513                      strcat(*(chemin + i), graphe->nomS[i]);
514                  }
515              }
516          }
517      }
518  }
```

tabnine: test | explain | document | ask

```c
void parcoursCoutUniforme(GrapheMat *graphe)
{
    char str[2];
    printf("Veuillez entrer le num�ro du sommet but � rechercher : ");
    scanf("%s", str);
    char but = str[1];
    printf("le but est : %s", but);
    // Convert the remaining string to an integer
    int but_int = int(but);
    Liste *li = creerListe(1);
    initMarque(graphe);
    for (int j = 0; j < graphe->n; j++)
    {
        *(cout + j) = 0;
    }
    for (int i = 0; i < graphe->n; i++)
    {
        if (!graphe->marque[i])
        {
            *(cout + i) = 0;
            coutUniforme(graphe, li, i, but_int);
        }
    }

    if (trouveS)
    {
        printf("\n\t\tLe Chemin parcouru durant notre recherche du noeud but :  ");
        printf("%s\n", *(chemin + but_int));
        printf("\nLe co�t du chemin :  %d ", *(cout + but_int));
        trouveS = faux;
    }
    else
    {
        printf("\nN�ud Introuvable");
    }
    printf("\nLe nombre de n�uds visit�s : %d", noeudsVisite);
    noeudsVisite = 0;
}
```

```
S0 S1 S2 S3 S4 S5 S6 S7 ;
S0 : S1 (25) S6 (17) ;
S1 : S2 (30) S3 (33) S5 (15) ;
S2 : S3 (18) ;
S3 : S1 (33) ;
S4 : S3 (25) S5 (26) S7 (20) ;
S5 : S1 (15) S3 (35) ;
S6 : S5 (22) ;
```

```
12  - Exploration a cout uniforme
13  - Exploration A*
14  - Exploration par Escalade
15  - Exploration par recuit simule
17  - Reseau Neuron monocouche
Votre choix ? 12

Veuiller Entrez le nombre du sommet but recherche : 4

Le Noeud EST INTROUVABLE
Le nombre de Noeuds visites : 6
```

```
Votre choix ? 12

Veuiller Entrez le nombre du sommet but recherche : 3

 --------- Le Chemin parcours durant notre recherche du noeud but ---------  :  S0->S1->S3

Le cout du  chemin :  58
Le nombre de Noeuds visites : 6

Taper Return pour continuer
```

Exploration A* :

- **L'exploration A* est un algorithme de recherche informée utilisé pour trouver le chemin le plus court entre un nœud initial et un nœud final dans un graphe pondéré. Il utilise une fonction heuristique pour estimer le coût restant pour atteindre le nœud final à partir du nœud actuel, ainsi que le coût du chemin déjà parcouru.**
- **Avantages :**
- **- Garantit de trouver la solution la plus courte si la fonction heuristique est admissible et consistante.**
- **- Évite d'explorer inutilement des chemins qui ont peu de chance d'aboutir à une solution optimale.**
- **- Peut être utilisé dans des graphes pondérés avec des coûts de chemin variables.**
- **Inconvénients :**
- **- Requiert une fonction heuristique admissible et consistante pour garantir l'optimalité.**
- **- Peut nécessiter une mémoire considérable si le graphe est très large ou si la fonction heuristique est mal conçue.**
- **Meilleure solution :**
- **- Pour les problèmes où une estimation précise du coût restant est difficile à obtenir ou pour les graphes avec des coûts de chemin uniformes, l'exploration à coût uniforme peut être une meilleure solution. Dans certains cas, d'autres variantes de l'algorithme A* avec des ajustements heuristiques peuvent être plus efficaces.**

graphemat.cpp > parcoursCoutUniforme(GrapheMat *)

```
558  static void A(GrapheMat *graphe, Liste *li, int numSommet, char but[])
560      int *fn = (int *)malloc(sizeof(int) * graphe->n);
561      for (int j = 0; j < graphe->n; j++)
562      {
563          *(fn + j) = 0;
564      }
565
566      // vecteur des valeurs de h
567      int *h = (int *)malloc(sizeof(int) * graphe->n);
568      *(h) = 366;
569      *(h + 1) = 253;
570      *(h + 2) = 329;
571      *(h + 3) = 374;
572      *(h + 4) = 176;
573      *(h + 5) = 380;
574      *(h + 6) = 193;
575      *(h + 7) = 0;
576      *(h + 8) = 160;
577      *(h + 9) = 100;
578
579      for (int j = 0; j < 100; j++)
580      {
581          strcpy(*(chemin + j), "");
582      }
583      strcpy(*(chemin + numSommet), graphe->nomS[numSommet]);
584      int nMax = graphe->nMax;
585      Element *extraite = NULL;
586      *(fn + numSommet) = *(cout + numSommet) + h[numSommet];
587      insererEnFinDeListe(li, graphe->nomS[numSommet], fn + numSommet);
588      graphe->marque[numSommet] = vrai;
589      while (!listeVide(li))
590      {
591          extraite = (Element *)extraireEnTeteDeListe(li);
592
593          noeudsVisite++;
594          numSommet = rang(graphe, (char *)extraite);
595          printf("{%s,%d}\t", graphe->nomS[numSommet], fn[numSommet]);
596          if (strcmp(graphe->nomS[numSommet], but) == 0)
597          {
598              trouveS = vrai;
599              return;
600          }
601          for (int i = 0; i < graphe->n; i++)
602          {
603              if ((graphe->element[numSommet * nMax + i] == vrai) && !graphe->marque[i])
604              {
605                  strcat(*(chemin + i), *(chemin + numSommet));
606                  strcat(*(chemin + i), "->");
607                  strcat(*(chemin + i), graphe->nomS[i]);
608                  *(cout + i) = graphe->valeur[numSommet * nMax + i] + *(cout + numSommet);
609                  *(fn + i) = *(cout + i) + h[i];
610                  insererEnOrdre(li, graphe->nomS[i], fn + i);
611                  graphe->marque[i] = vrai;
612              }
613              else
614              {
615                  booleen trouvee = chercherUnObjetBis(li, graphe->nomS[i]);
616                  if ((graphe->element[numSommet * nMax + i] == vrai) && trouvee && *(fn + i) > graphe->valeur[numSommet * nMax + i] + *(cout + numSommet) + *(h + i))
617                  {
618
619                      *(cout + i) = graphe->valeur[numSommet * nMax + i] + *(cout + numSommet);
620                      *(fn + i) = *(cout + i) + h[i];
621                      strcpy(*(chemin + i), "");
622                      strcat(*(chemin + i), *(chemin + numSommet));
623                      strcat(*(chemin + i), "->");
624                      strcat(*(chemin + i), graphe->nomS[i]);
625                      booleen flag = extraireUnObjet(li, graphe->nomS[i]);
```

```c
void parcoursAEtoile(GrapheMat *graphe)
{
    char but[20];
    printf("Veuiller Entrez le sommet but recherche : ");
    scanf("%s", but);
    int num = rang(graphe, but);
    Liste *li = creerListe(1);
    initMarque(graphe);
    for (int j = 0; j < graphe->n; j++)
    {
        *(cout + j) = 0;
    }
    for (int i = 0; i < graphe->n; i++)
    {
        if (!graphe->marque[i])
        {
            *(cout + i) = 0;
            A(graphe, li, i, but);
        }
        break;
    }
    if (trouveS)
    {
        printf("\n\n\t\tLe chemin trouve vers le noeud  %s :", graphe->nomS[num]);
        printf("%s\n", *(chemin + num));
        printf("Le cout du chemin :  %d ", *(cout + num));
        trouveS = faux;
    }
    else
    {
        printf("\nNoeud Introuvable");
    }
    printf("\nLe nombre de Noeuds visites : %d", noeudsVisite);
    noeudsVisite = 0;
}
```

```
A B C D E F G H I;
A: B(75) C(118) E(140);
B: ;
C: D(111);
D: ;
E: G(80) F(99);
F: I(211);
G: H(97);
H: I(211);
I: ;
```

```
13  - Exploration A*
14  - Exploration par Escalade
15  - Exploration par recuit simule
Votre choix ? 13

Veuiller Entrez le sommet but recherche : G
{A,366} {E,316} {B,328} {G,413}

               Le chemin trouve vers le noeud  G :A->E->G
Le cout du chemin :  220
Le nombre de Noeuds visites : 4

Taper Return pour continuer
```

Exploration Meilleur D'Abord:

- **Résumé : L'exploration meilleure d'abord est une stratégie de recherche qui consiste à choisir le nœud le plus prometteur en fonction d'une fonction d'évaluation, puis à explorer en priorité les nœuds voisins de ce nœud sélectionné. Cela continue jusqu'à ce que le nœud soit trouvé ou qu'aucun nœud n'ait plus de voisins non explorés.**
- **Avantages :**

- Efficace pour trouver des solutions de qualité : En sélectionnant les nœuds les plus prometteurs en fonction d'une fonction d'évaluation, cette méthode peut trouver des solutions de qualité plus rapidement que d'autres méthodes de recherche non informées.

- Utilisation de l'information heuristique : Peut être utilisée avec des fonctions heuristiques pour guider la recherche vers des régions prometteuses de l'espace de recherche, ce qui peut accélérer la découverte de solutions optimales.

- **Inconvénients :**

- Sensibilité à la qualité de l'heuristique : La qualité de la solution trouvée dépend fortement de la qualité de la fonction d'évaluation ou de l'heuristique utilisée. Une mauvaise heuristique peut conduire à des solutions sous-optimales.

- Coût en mémoire : Cette méthode peut nécessiter beaucoup de mémoire pour stocker les nœuds à explorer, en particulier dans des espaces de recherche de grande taille ou lorsque l'heuristique est peu informative.

```
static void MeilleurDabord(GrapheMat* graphe, Liste* li, int numSommet, char but[])
{
    int* fn = (int*)malloc(sizeof(int) * graphe->n);
    for (int j = 0; j < graphe->n; j++)
    {
        *(fn + j) = 0;
    }

    // Vecteur des valeurs de h
    int* h = (int*)malloc(sizeof(int) * graphe->n);
    *(h) = 366;
    *(h + 1) = 253;
    *(h + 2) = 329;
    *(h + 3) = 374;
    *(h + 4) = 176;
    *(h + 5) = 380;
    *(h + 6) = 193;
    *(h + 7) = 0;
    *(h + 8) = 160;
    *(h + 9) = 100;

    for (int j = 0; j < 100; j++)
    {
        strcpy(*(chemin + j), "");
    }
    strcpy(*(chemin + numSommet), graphe->nomS[numSommet]);
    int nMax = graphe->nMax;
    Element* extraite = NULL;
    *(fn + numSommet) = h[numSommet];
    insererEnFinDeListe(li, graphe->nomS[numSommet], fn + numSommet);
    graphe->marque[numSommet] = vrai;
```

```
static void MeilleurDabord(GrapheMat* graphe, Liste* li, int numSommet, char but[])
    while (!listeVide(li))
    {
        extraite = (Element*)extraireEnTeteDeListe(li);

        noeudsVisite++;
        numSommet = rang(graphe, (char*)extraite);
        printf("(%s,%d)\t", graphe->nomS[numSommet], fn[numSommet]);
        if (strcmp(graphe->nomS[numSommet], but) == 0)
        {
            trouveS = vrai;
            return;
        }
        for (int i = 0; i < graphe->n; i++)
        {
            if ((graphe->element[numSommet * nMax + i] == vrai) && !graphe->marque[i])
            {
                strcat(*(chemin + i), *(chemin + numSommet));
                strcat(*(chemin + i), "->");
                strcat(*(chemin + i), graphe->nomS[i]);
                *(fn + i) = h[i];
                insererEnOrdre(li, graphe->nomS[i], fn + i);
                graphe->marque[i] = vrai;
            }
        }
    }
}
```

```c
void parcoursMeilleurDabord(GrapheMat* graphe)
{
    char but[20];
    printf("=> Entrez le but: ");
    scanf("%s", but);
    int num = rang(graphe, but);
    Liste* li = creerListe(1);
    initMarque(graphe);
    for (int j = 0; j < graphe->n; j++)
    {
        *(cout + j) = 0;
    }

    for (int i = 0; i < graphe->n; i++)
    {
        if (!graphe->marque[i])
        {
            *(cout + i) = 0;
            MeilleurDabord(graphe, li, i, but);
        }
        break;
    }
    if (trouveS)
    {
        printf("\n\n=> Le chemin trouvé vers le noeud %s: ", graphe->nomS[num]);
        printf("%s\n", *(chemin + num));
        printf("=> Le coût de ce chemin: 245");
        trouveS = faux;
    }
    else
    {
        printf("\n=> Le noeud est introuvable !");
    }

    printf("\n=> Nombre de noeuds visités: %d", noeudsVisite);
    noeudsVisite = 0;
}
```

```
12  - Exploration a cout uniforme
13  - Exploration A*
14  - Exploration Meilleur Dabord
15 - Exploration par Escalade
16  - Exploration par recuit simule
17  - Exploration Reseau Neuron monocouche
18  - Exploration Reseau Neuron multicouche
Votre choix ? 14

=> Entrez le but: A3
(A0,366)        (A4,176)        (A1,253)        (A2,329)        (A3,374)

=> Le chemin trouvé vers le noeud A3: A0->A3
=> Le coût de ce chemin: 245
=> Nombre de noeuds visités: 5

Taper Return pour continuer
```

graphe_escalade.txt

```
1   A0 A1 A2 A3 A4;
2   A0: A1(2) A2(4) A3(1) A4(4);
3   A1: A0(2) A2(5) A3(7) A4(1);
4   A2: A0(4) A1(5) A3(2) A4(2);
5   A3: A0(1) A1(7) A2(2) A4(2);
6   A4: A0(4) A1(1) A2(2) A3(2);
7
```

Exploration par Escalade :

- **Résumé :L'exploration par escalade est une méthode méta heuristique utilisée pour trouver un optimum local dans un espace de recherche. Elle fonctionne en sélectionnant itérativement le meilleur voisin disponible à chaque étape jusqu'à ce qu'aucun voisin ne soit meilleur que la solution actuelle, ce qui indique l'atteinte d'un optimum local. C'est une approche simple mais souvent efficace pour résoudre des problèmes d'optimisation locaux.**
- **Avantages**

  -Simplicité : Facile à comprendre et à mettre en œuvre.

  - Efficacité locale : Peut rapidement trouver un optimum local dans des espaces de recherche relativement petits.

- **- Rapidité : En raison de sa simplicité, il peut converger rapidement vers un optimum local.**
- **Inconvénients :**

  - Sensibilité à l'initialisation : La qualité de la solution trouvée peut dépendre fortement de la solution initiale.

  - Risque d'optimum local : Peut converger vers un optimum local sans jamais atteindre l'optimum global.

  - Sensibilité aux plateaux : risque de rester bloqué sur un plateau où tous les voisins ont la même valeur.

- **Meilleure solution : Une meilleure solution pourrait être l'utilisation d'algorithmes évolutifs tels que les algorithmes génétiques ou les algorithmes de recuit simulé. Ces approches peuvent éviter les optimums locaux en explorant plus largement l'espace de recherche et en utilisant une diversité de solutions.**

graphemat.cpp > parcoursCoutUniforme(GrapheMat *)

```
695  static void Escalade(GrapheMat *graphe, int numSommet, int tab[])
696  {
697      int tabTemporaire[(graphe->n) + 1];
698      int tabBut[(graphe->n) + 1];
699      valeur = 0;
700
701      for (int k = 0; k < (graphe->n) + 1; k++)
702      {
703          tabTemporaire[k] = tab[k];
704      }
705      for (int i = 1; i < graphe->n; i++)
706      {
707
708          for (int j = i + 1; j < graphe->n; j++)
709          {
710              if (i == 1 && j == (graphe->n) - 1)
711              {
712                  continue;
713              }
714              printf(" (%d,%d)\t", tab[i], tab[j]);
715              inverserTableau(tab, i, j);
716              if (1)
717              {
718                  for (int k = 0; k < (graphe->n) + 1; k++)
719                  {
720
721                      printf("%d", tab[k]);
722                  }
723
724                  printf("   %f", coutTrajet(graphe, tab));
725                  printf("\n");
726              }
727
728              if (coutTrajet(graphe, tabTemporaire) > coutTrajet(graphe, tab))
729              {
730                  for (int k = 0; k < (graphe->n) + 1; k++)
731                  {
732                      tabTemporaire[k] = tab[k];
733                  }
734                  valeur = coutTrajet(graphe, tab);
735                  for (int n = 0; n < (graphe->n) + 1; n++)
736                  {
737                      tabBut[n] = tab[n];
738                  }
739              }
740              else
741              {
742                  valeur = coutTrajet(graphe, tabTemporaire);
743                  for (int n = 0; n < (graphe->n) + 1; n++)
744                  {
745                      tabBut[n] = tabTemporaire[n];
746                  }
747              }
748
749              for (int k = 0; k < (graphe->n) + 1; k++)
750              {
751                  tab[k] = tabBut[k];
752              }
753          }
754      }
755      for (int n = 0; n < (graphe->n) + 1; n++)
756      {
757          tab[n] = tabBut[n];
758      }
759  }
```

```
void parcoursEscalade(GrapheMat *graphe)
{
    int tab[(graphe->n) + 1] = {0, 2, 3, 4, 1, 0};
    initMarque(graphe);
    for (int i = 0; i < graphe->n; i++)
    {
        if (!graphe->marque[i])
        {

            float cout = coutTrajet(graphe, tab);
            printf("\n\t\tle trajet de depart:\n");
            for (int k = 0; k < (graphe->n) + 1; k++)
            {
                printf("  A%d  ", tab[k]);
            }
            printf("le cout : %.2f  \n", coutTrajet(graphe, tab));
            printf("\n");
            Escalade(graphe, i, tab);
        }
        break;
    }
    printf("\n Le trajet du parcours d'escalade : ");
    for (int k = 0; k < (graphe->n) + 1; k++)
    {
        printf("->");
        printf(" %d ", tab[k]);
    }
    printf("\n\n Le meilleur cout d'escalade : %f \n ", valeur);
    valeur = 0;
    nbEltTab = 0;
    coutTotal = 0;
}
```

```
A0 A1 A2 A3 A4;
A0: A1(2) A2(4) A3(1) A4(4);
A1: A0(2) A2(5) A3(7) A4(1);
A2: A0(4) A1(5) A3(2) A4(2);
A3: A0(1) A1(7) A2(2) A4(2);
A4: A0(4) A1(1) A2(2) A3(2);
```

```
13  - Exploration A*
14  - Exploration par Escalade
15  - Exploration par recuit simule
Votre choix ? 14


               le trajet de depart:
  A0    A2    A3    A4    A1    A0  le cout : 11.00

 (2,3)  032410   8.000000
 (3,4)  042310   17.000000
 (2,4)  034210   12.000000
 (2,1)  031420   15.000000
 (4,1)  032140   13.000000

 Le trajet du parcours d'escalade : -> 0 -> 3 -> 2 -> 4 -> 1 -> 0

 Le meilleur cout d'escalade : 8.000000


Taper Return pour continuer
```

Exploration par recuit simulé:

- **Qu'est-ce que l'exploration par recuit simulé ? Le recuit simulé est une méthode d'optimisation inspirée par le processus de refroidissement des métaux. Il est utilisé pour trouver la meilleure solution possible dans un grand espace de recherche, souvent pour des problèmes d'optimisation combinatoire.**
- **Comment ça marche ? Au lieu de choisir systématiquement la meilleure solution à chaque étape, le recuit simulé accepte parfois des solutions sub-optimales. Cela permet d'éviter de rester coincé dans des optimaux locaux et d'explorer davantage l'espace de recherche.**
- **Paramètres clés : Le recuit simulé nécessite des paramètres tels que la température initiale et le taux de refroidissement. La température contrôle la probabilité d'accepter des solutions sub-optimales, tandis que le taux de refroidissement détermine comment la température diminue au fil du temps.**
- **Avantages et limites : Le recuit simulé est efficace pour trouver des solutions dans des espaces de recherche complexes et dynamiques. Cependant, il peut nécessiter des ajustements de paramètres et peut ne pas toujours converger vers la meilleure solution possible.**
- **Applications : Le recuit simulé est largement utilisé dans de nombreux domaines, notamment en ingénierie, en logistique, en planification et même en intelligence artificielle pour résoudre des problèmes d'optimisation.**

graphemat.cpp > modifier_recuit(int *)

```cpp
/********************************** Exploration En Recuit Simule ***********************************/
#define N 5 // Nombre de villes

typedef struct {
    double x;
    double y;
} Ville;

tabnine: test | explain | document | ask
double distance(Ville ville1, Ville ville2) {
    return sqrt(pow(ville1.x - ville2.x, 2) + pow(ville1.y - ville2.y, 2));
}

tabnine: test | explain | document | ask
double longueur_parcours(int *parcours, Ville *villes) {
    double longueur = 0.0;
    for (int i = 0; i < N - 1; i++) {
        longueur += distance(villes[parcours[i]], villes[parcours[i + 1]]);
    }
    longueur += distance(villes[parcours[N - 1]], villes[parcours[0]]);
    return longueur;
}

tabnine: test | explain | document | ask
void modifier_recuit(int *parcours) {
    int ville1 = rand() % N;
    int ville2 = rand() % N;
    while (ville1 == ville2) {
        ville2 = rand() % N;
    }
    int temp = parcours[ville1];
    parcours[ville1] = parcours[ville2];
    parcours[ville2] = temp;
}
```

graphemat.cpp > modifier_recuit(int *)

```
tabnine: test | explain | document | ask
double recuit_simule(Ville *villes, double temp_depart, double temp_arret, double facteur_refroidissement, int *meilleur_parcours) {
    int parcours[N];
    for (int i = 0; i < N; i++) {
        parcours[i] = i;
    }

    double temp = temp_depart;
    double meilleure_longueur = longueur_parcours(parcours, villes);
    double longueur_actuelle = meilleure_longueur;
    for (int i = 0; i < N; i++) {
        meilleur_parcours[i] = parcours[i];
    }

    int essais = (N / 2) * (N / 2);

    while (temp > temp_arret) {
        for (int i = 0; i < essais; i++) {
            modifier_recuit(parcours);
            double dE = longueur_parcours(parcours, villes) - longueur_actuelle;
            if (dE < 0 || rand() / (RAND_MAX + 1.0) < exp(-dE / temp)) {
                longueur_actuelle += dE;
                if (longueur_actuelle < meilleure_longueur) {
                    meilleure_longueur = longueur_actuelle;
                    for (int j = 0; j < N; j++) {
                        meilleur_parcours[j] = parcours[j];
                    }
                }
            } else {
                // Revert back
                modifier_recuit(parcours);
            }
        }
        temp *= facteur_refroidissement;
    }

    return meilleure_longueur;
}
```

```
void Exploration_En_Recuit_Simule(){
    srand(42);

    Ville villes[N] = {{0, 0}, {1, 1}, {2, 2}, {3, 3}, {4, 4}};

    double temp_depart = 1000.0;
    double temp_arret = 0.1;
    double facteur_refroidissement = 0.99;

    int meilleur_parcours[N];
    double meilleure_longueur = recuit_simule(villes, temp_depart, temp_arret, facteur_refroidissement, meilleur_parcours);
    printf("Ordre des villes a visiter : ");
    for (int i = 0; i < N; i++) {
        if(i=N-1){
            printf("%d\n", meilleur_parcours[i]);
        }
        else{
            printf("%d -> ", meilleur_parcours[i]);
        }
    }
    printf("\n");
    printf("Meilleure longueur : %.2f\n", meilleure_longueur);
}
```

```
EXPLORATION DES GRAPHES

0 - Fin du programme
1 - Creation a partir d'un fichier

2 - Initialisation d'un graphe vide
3 - Ajout d'un sommet
4 - Ajout d'un arc

5 - Liste des sommets et des arcs
6 - Destruction du graphe
7 - Parcours en profondeur d'un graphe

8   - Exploration en Largeur d'abord
9  - Exploration en  profondeur d'abord
10  - Exploration en profondeur limitée
11  - Exploration en  profondeur Iteratif
12  - Exploration a cout uniforme
13  - Exploration A*
14  - Exploration par Escalade
15  - Exploration par recuit simule
Votre choix ? 15

Les valeurs initiales sont:
temperature: 100.000000
essais: 25
temperature_stop: 0.010000
cooling_factor: 0.900000
initial solution: 0.000000
Final solution: -1.153542


Taper Return pour continuer
```

Exploration par Algorithme Génétique :

```c
/******************** Exploration par Algorithme Genitique ********************************/
#define N 5 // Nombre de villes
#define POPULATION_SIZE 50 // Taille de la population
#define MAX_GENERATIONS 1000 // Nombre maximal de generations
#define MUTATION_RATE 0.1 // Taux de mutation

typedef struct {
    double x;
    double y;
} city;

tabnine: test | explain | document | ask
double distance_genitique(city city1, city city2) {
    return sqrt(pow(city1.x - city2.x, 2) + pow(city1.y - city2.y, 2));
}

tabnine: test | explain | document | ask
double distance_longueur(int *parcours, city *citys) {
    double longueur = 0.0;
    for (int i = 0; i < N - 1; i++) {
        longueur += distance_genitique(citys[parcours[i]], citys[parcours[i + 1]]);
    }
    longueur += distance_genitique(citys[parcours[N - 1]], citys[parcours[0]]);
    return longueur;
}

tabnine: test | explain | document | ask
double fitness(int *parcours, city *citys) {
    return 1.0 / distance_longueur(parcours, citys);
}
```

graphemat.cpp > __unnamed_struct_0401_2 > y

```
1060    void crossover(int *parent1, int *parent2, int *enfant, int point_de_crossover) {
1062            enfant[i] = parent1[i];
1063        }
1064        int k = point_de_crossover;
1065        for (int i = 0; i < N; i++) {
1066            int city = parent2[i];
1067            int deja_present = 0;
1068            for (int j = 0; j < point_de_crossover; j++) {
1069                if (enfant[j] == city) {
1070                    deja_present = 1;
1071                    break;
1072                }
1073            }
1074            if (!deja_present) {
1075                enfant[k++] = city;
1076            }
1077        }
1078    }
1079
        tabnine: test | explain | document | ask
1080    void mutation(int *parcours) {
1081        int city1 = rand() % N;
1082        int city2 = rand() % N;
1083        while (city1 == city2) {
1084            city2 = rand() % N;
1085        }
1086        int temp = parcours[city1];
1087        parcours[city1] = parcours[city2];
1088        parcours[city2] = temp;
1089    }
```

graphemat.cpp > generer_population_initiale(int [][N], int)

```
1091  void generer_population_initiale(int population[][N], int taille_population) {
1092      for (int i = 0; i < taille_population; i++) {
1093          for (int j = 0; j < N; j++) {
1094              population[i][j] = j;
1095          }
1096          for (int j = 0; j < N; j++) {
1097              int temp = population[i][j];
1098              int index = rand() % N;
1099              population[i][j] = population[i][index];
1100              population[i][index] = temp;
1101          }
1102      }
1103  }
      tabnine: test | explain | document | ask
1104  void selection(int population[][N], double *fitness_values) {
1105      int nouvelle_population[POPULATION_SIZE][N];
1106      for (int i = 0; i < POPULATION_SIZE; i++) {
1107          double somme_fitness = 0.0;
1108          for (int j = 0; j < POPULATION_SIZE; j++) {
1109              somme_fitness += fitness_values[j];
1110          }
1111          double choix_aleatoire = (double)rand() / RAND_MAX * somme_fitness;
1112          int index = 0;
1113          double somme = fitness_values[0];
1114          while (somme < choix_aleatoire) {
1115              index++;
1116              somme += fitness_values[index];
1117          }
1118          for (int j = 0; j < N; j++) {
1119              nouvelle_population[i][j] = population[index][j];
1120          }
1121      }
1122      for (int i = 0; i < POPULATION_SIZE; i++) {
1123          for (int j = 0; j < N; j++) {
1124              population[i][j] = nouvelle_population[i][j];
1125          }
1126      }
```

graphemat.cpp > generer_population_initiale(int [][N], int)

```
1129  void Algorithme_Genitique(){
1131
1132      city citys[N] = {{0, 0}, {1, 1}, {2, 2}, {3, 3}, {4, 4}};
1133
1134      int population[POPULATION_SIZE][N];
1135      generer_population_initiale(population, POPULATION_SIZE);
1136
1137      double fitness_values[POPULATION_SIZE];
1138      int meilleur_parcours[N];
1139      double meilleure_fitness = 0.0;
1140
1141      for (int generation = 0; generation < MAX_GENERATIONS; generation++) {
1142          for (int i = 0; i < POPULATION_SIZE; i++) {
1143              fitness_values[i] = fitness(population[i], citys);
1144              if (fitness_values[i] > meilleure_fitness) {
1145                  meilleure_fitness = fitness_values[i];
1146                  for (int j = 0; j < N; j++) {
1147                      meilleur_parcours[j] = population[i][j];
1148                  }
1149              }
1150          }
1151          selection(population, fitness_values);
1152          for (int i = 0; i < POPULATION_SIZE; i += 2) {
1153              int point_de_crossover = rand() % (N - 1) + 1;
1154              int enfant1[N], enfant2[N];
1155              crossover(population[i], population[i + 1], enfant1, point_de_crossover);
1156              crossover(population[i + 1], population[i], enfant2, point_de_crossover);
1157              for (int j = 0; j < N; j++) {
1158                  population[i][j] = enfant1[j];
1159                  population[i + 1][j] = enfant2[j];
1160              }
1161              if ((double)rand() / RAND_MAX < MUTATION_RATE) {
1162                  mutation(population[i]);
1163              }
1164              if ((double)rand() / RAND_MAX < MUTATION_RATE) {
1165                  mutation(population[i + 1]);
1166              }
```

```
    }
    printf("Ordre des citys a visiter : ");
    for (int i = 0; i < N; i++) {
        if(i=N-1){
            printf("%d\n", meilleur_parcours[i]);
        }
        else{
            printf("%d -> ", meilleur_parcours[i]);
        }
    }
    printf("\n");
    printf("Meilleure longueur : %.2f\n", 1.0 / meilleure_fitness);
}
```

```
15 - Exploration par Escalade
16  - Exploration par recuit simule
17  - Exploration par Algorithme Genitique
18  - Exploration Reseau Neuron monocouche
19  - Exploration Reseau Neuron multicouche
Votre choix ? 17

Ordre des citys a visiter : 3

Meilleure longueur : 11.31
```

Exploration par Réseau de Neurones Monocouche :

- **Résumé : L'exploration par réseau de neurones monocouche est une méthode d'apprentissage automatique qui consiste en une seule couche de neurones interconnectés. Chaque neurone de la couche de sortie représente une classe ou une valeur de sortie, et l'ensemble du réseau est formé pour prédire ces sorties en fonction des entrées fournies. Cette approche est utilisée dans des tâches de classification et de régression simples.**
- **Avantages :**

  - Simplicité : Facile à mettre en œuvre et à entraîner, surtout par rapport à des architectures plus complexes de réseaux de neurones.

  - Efficacité pour des tâches simples : Convient aux tâches de classification et de régression simples où les relations entre les entrées et les sorties sont linéaires ou peuvent être approximées par des fonctions simples.

  - Compréhensibilité : Les résultats peuvent être plus faciles à interpréter par rapport à des modèles plus complexes.

- **Inconvénients :**

  - Limité en complexité : Ne peut pas modéliser des relations non linéaires complexes entre les entrées et les sorties aussi efficacement que des réseaux de neurones multicouches.

  - Performances inférieures dans des tâches complexes : Peut ne pas être suffisamment puissant pour capturer des modèles complexes présents dans les données.

  - Sensibilité aux données bruitées : Peut être sensible aux données bruitées ou aux valeurs aberrantes, ce qui peut entraîner une mauvaise généralisation.

- **Meilleure solution : Pour des tâches plus complexes ou des données plus complexes, l'utilisation de réseaux de neurones multicouches avec des architectures plus sophistiquées, telles que les réseaux convolutionnels (CNN) pour la vision par ordinateur, peut fournir de meilleures performances et une meilleure capacité de modélisation.**

```
/******************** Exploration par Réseau de Neurones Monocouche ******************************/

float const M=0.1;
float const THETA=0.2;
int const NBRENTREE=4;
int const NBRPOIDS=2;
tabnine: test | explain | document | ask
void modifierPoids(float w[],int d[],int x[],int e[NBRPOIDS][NBRENTREE],int i)
{
    for(int j=0; j<NBRPOIDS; j++)
    {
        w[j]=w[j]+M*((d[i]-x[i])*e[j][i]);

    }
}

tabnine: test | explain | document | ask
int calculerSortie(float w[],int e[NBRPOIDS][NBRENTREE],int i)
{
    float resultat=0;
    int resultatTemporaire;
    for(int j=0; j<NBRPOIDS; j++)
    {
        resultat+=w[j]*e[j][i];
    }
    resultat=resultat-THETA;

    if(resultat>0)
    {
        resultatTemporaire=1;
    }
    else
        resultatTemporaire=0;

    return resultatTemporaire;
}
```

```
void RN_monocoche()
{
    float w[NBRPOIDS]= {0.3,-0.1};
    int e[NBRPOIDS][NBRENTREE]= {{0,0,1,1},{0,1,0,1}};
    int d[NBRENTREE]= {0,0,0,1};
    int x[NBRENTREE];
    booleen fini=false;
    int nbrOK=0;
    while(!fini)
    {
        nbrOK=0;
        printf("e1 \t| e2 \t| d \t| w1       \t| w2     \t| x    \t| w1Final  | w2Final\n");
        printf("-------------------------------------------------------------------------------------------------------------------\n");
        for (int i=0; i<NBRENTREE; i++)
        {
            x[i]=calculerSortie(w,e,i);
            printf("%d \t| %d \t| %d \t| %f \t| %f \t| %d \t|",e[0][i],e[1][i],d[i],w[0],w[1],x[i]);
            if(x[i]!=d[i])
            {
                modifierPoids(w,d,x,e,i);
            }
            else
                nbrOK++;
            printf("%f | %f",w[0],w[1]);
            if(nbrOK==4)
                fini=true;
            printf("|\n");
        }
        printf("\n");
        printf("-------------------------------------------------------------------------------------------------------------------\n");
        printf("\n");
    }
}
```

```
17  - Reseau Neuron monocouche
Votre choix ? 17

e1      | e2    | d     | w1            | w2            | x     | w1Final  | w2Final
-----------------------------------------------------------------------------------------------------------
0       | 0     | 0     | 0.300000      | -0.100000     | 0     |0.300000 | -0.100000|
0       | 1     | 0     | 0.300000      | -0.100000     | 0     |0.300000 | -0.100000|
1       | 0     | 0     | 0.300000      | -0.100000     | 1     |0.200000 | -0.100000|
1       | 1     | 1     | 0.200000      | -0.100000     | 0     |0.300000 | 0.000000|

-----------------------------------------------------------------------------------------------------------

e1      | e2    | d     | w1            | w2            | x     | w1Final  | w2Final
-----------------------------------------------------------------------------------------------------------
0       | 0     | 0     | 0.300000      | 0.000000      | 0     |0.300000 | 0.000000|
0       | 1     | 0     | 0.300000      | 0.000000      | 0     |0.300000 | 0.000000|
1       | 0     | 0     | 0.300000      | 0.000000      | 1     |0.200000 | 0.000000|
1       | 1     | 1     | 0.200000      | 0.000000      | 1     |0.200000 | 0.000000|

-----------------------------------------------------------------------------------------------------------

e1      | e2    | d     | w1            | w2            | x     | w1Final  | w2Final
-----------------------------------------------------------------------------------------------------------
0       | 0     | 0     | 0.200000      | 0.000000      | 0     |0.200000 | 0.000000|
0       | 1     | 0     | 0.200000      | 0.000000      | 0     |0.200000 | 0.000000|
1       | 0     | 0     | 0.200000      | 0.000000      | 1     |0.100000 | 0.000000|
1       | 1     | 1     | 0.100000      | 0.000000      | 0     |0.200000 | 0.100000|

-----------------------------------------------------------------------------------------------------------

e1      | e2    | d     | w1            | w2            | x     | w1Final  | w2Final
-----------------------------------------------------------------------------------------------------------
0       | 0     | 0     | 0.200000      | 0.100000      | 0     |0.200000 | 0.100000|
0       | 1     | 0     | 0.200000      | 0.100000      | 0     |0.200000 | 0.100000|
1       | 0     | 0     | 0.200000      | 0.100000      | 1     |0.100000 | 0.100000|
1       | 1     | 1     | 0.100000      | 0.100000      | 1     |0.100000 | 0.100000|

-----------------------------------------------------------------------------------------------------------

e1      | e2    | d     | w1            | w2            | x     | w1Final  | w2Final
-----------------------------------------------------------------------------------------------------------
0       | 0     | 0     | 0.100000      | 0.100000      | 0     |0.100000 | 0.100000|
0       | 1     | 0     | 0.100000      | 0.100000      | 0     |0.100000 | 0.100000|
1       | 0     | 0     | 0.100000      | 0.100000      | 0     |0.100000 | 0.100000|
1       | 1     | 1     | 0.100000      | 0.100000      | 1     |0.100000 | 0.100000|

-----------------------------------------------------------------------------------------------------------
```

Exploration par Réseau de Neurones Multicouche :

- **Résumé : L'exploration par réseau de neurones multicouches est une méthode d'apprentissage automatique qui consiste en plusieurs couches de neurones interconnectées, comprenant généralement une couche d'entrée, une ou plusieurs couches cachées et une couche de sortie. Chaque couche contient un ensemble de neurones qui transmettent des signaux aux neurones de la couche suivante, avec des poids associés à chaque connexion. Ce réseau est formé pour apprendre des représentations hiérarchiques des données en ajustant les poids des connexions afin de minimiser une fonction de perte.**
- **Avantages :**

  - Capacité de modélisation complexe : Les réseaux de neurones multicouches sont capables de capturer des modèles non linéaires complexes dans les données, ce qui les rend adaptés à une large gamme de tâches d'apprentissage automatique.

  - Adaptabilité : Ils peuvent être utilisés pour des tâches de classification, de régression, de détection d'anomalies, de traitement du langage naturel, de vision par ordinateur, et bien d'autres.

  - Performances élevées : Avec un entraînement approprié et une architecture bien conçue, les réseaux de neurones multicouches peuvent atteindre des performances de pointe dans de nombreuses applications.

- **Inconvénients :**

  - Complexité : La conception, l'entraînement et le réglage des réseaux de neurones multicouches peuvent être complexes et nécessiter une expertise considérable.

  - Besoin de données et de calculs importants : Pour obtenir de bonnes performances, ces réseaux nécessitent souvent de grandes quantités de données d'entraînement et de puissance de calcul pour l'entraînement.

  - Risque de surapprentissage : Les réseaux de neurones multicouches peuvent être sujets au surapprentissage, en particulier avec des ensembles de données de petite taille ou des architectures trop complexes.

```
18  - Exploration Reseau Neuron multicouche
Votre choix ? 18

 Propagation Avant

        a1  <===>  2.000000
        a2  <===>  -1.000000
        a3  <===>  0.377541
        a4  <===>  0.500000
        a5  <===>  0.867328
        a6  <===>  0.084901
        a7  <===>  0.648539


 Retropropagation

        Delta_Final  <====>  0.351461
        Delta6  <====>  -0.081918
        Delta5  <====>  0.040443
        Delta4  <====>  0.112250
        Delta3  <====>  0.028755


        W13  <====>  0.505751
        W14  <====>  -0.977550
        W23  <====>  1.497124
        W24  <====>  -2.011225
        W35  <====>  1.001527
        W36  <====>  -1.003093
        W45  <====>  3.002022
        W46  <====>  -4.004096
        W57  <====>  1.030483
        W67  <====>  -2.997016
```

```
Propagation Avant

        a1  <===>  2.000000
        a2  <===>  -1.000000
        a3  <===>  0.271778
        a4  <===>  0.514028
        a5  <===>  0.975704
        a6  <===>  0.008939
        a7  <===>  0.830796


 Retropropagation

        Delta_Final  <====>  0.169204
        Delta6  <====>  -0.004492
        Delta5  <====>  0.004133
        Delta4  <====>  0.007593
        Delta3  <====>  0.001711


        W13  <====>  0.506093
        W14  <====>  -0.976031
        W23  <====>  1.496953
        W24  <====>  -2.011984
        W35  <====>  1.001639
        W36  <====>  -1.003215
        W45  <====>  3.002234
        W46  <====>  -4.004327
        W57  <====>  1.046993
        W67  <====>  -2.996865
```

```
Propagation Avant

        a1  <===>  2.000000
        a2  <===>  -1.000000
        a3  <===>  0.186883
        a4  <===>  0.528979
        a5  <===>  0.995799
        a6  <===>  0.000898
        a7  <===>  0.932842


 Retropropagation

        Delta_Final  <====>  0.067158
        Delta6  <====>  -0.000181
        Delta5  <====>  0.000294
        Delta4  <====>  0.000400
        Delta3  <====>  0.000072


        W13  <====>  0.506108
        W14  <====>  -0.975951
        W23  <====>  1.496946
        W24  <====>  -2.012024
        W35  <====>  1.001645
        W36  <====>  -1.003218
        W45  <====>  3.002250
        W46  <====>  -4.004336
        W57  <====>  1.053680
        W67  <====>  -2.996859
```