Concours commun Mines-Ponts, France

Session 2026, filière MPI

Informatique 2

Durée : 4 heures

Ce que traite le sujet

- Section 1 : représentation relationnelle des données
- Section 2 : représentation en mémoire et navigation dans le monde
- Section 3 : recherche rapide d'un chemin avec heuristique
- Section 4 : arbres BSP et dérécursivation

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/mines/mpi/2026/info2

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

A2026 – INFO II MPI

ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA,
TÉLÉCOM PARIS, MINES PARIS - PSL,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2026

DEUXIÈME ÉPREUVE D'INFORMATIQUE

Durée de l'épreuve : 4 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

Les candidats sont priés de mentionner de façon apparente sur la première page de la copie :

INFORMATIQUE II - MPI

Cette épreuve concerne uniquement les candidats de la filière MPI.

L'énoncé de cette épreuve comporte 17 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines-Ponts.

Regards contemporains sur des algorithmes des années 1990

Conception et optimisation pour moteurs de jeux vidéo

Préliminaires

L'épreuve a pour objectif d'étudier la conception et l'analyse d'algorithmes utilisés dans les moteurs de jeux vidéo dits rétros, développés dans les années 1990. À cette époque, les fortes contraintes de performances et de capacité mémoire des plateformes matérielles imposaient des choix algorithmiques et des structures de données particulièrement optimisés.

En s'appuyant sur le monde d'un jeu vidéo, stocké initialement sous forme relationnelle puis manipulé en mémoire, on s'intéressera successivement aux problèmes de représentation des données, de recherche de chemins et de visibilité. L'accent sera mis sur l'adéquation entre les algorithmes étudiés, les structures de données employées et les contraintes matérielles considérées.

L'épreuve est composée de plusieurs parties largement indépendantes, mais reposant sur des notions et des objets communs. Elle comporte au total 38 questions.

Travail attendu

Pour répondre à une question, il est toujours possible d'admettre et d'utiliser les résultats attendus dans les questions précédentes, et ce même si elles n'ont pas été abordées.

Le langage C constitue le langage attendu pour le traitement des questions de programmation. Les questions portant sur les bases de données relationnelles utilisent le langage SQL pour l'écriture de la requête.

Une attention particulière sera portée à la correction, à la lisibilité et à la structuration du code proposé. Dans le cadre d'une épreuve réalisée sur support papier, une tolérance raisonnable pourra être envisagée sur certains aspects purement syntaxiques (par exemple l'oubli d'un « ; »). En revanche, un code difficilement lisible, insuffisamment structuré, clairement trop long, ou faisant appel à des constructions non conformes aux standards du langage C — notamment des constructions empruntées à d'autres langages — sera sanctionné.

Un soin particulier devra être apporté à la gestion de la mémoire ainsi qu'à la bonne initialisation des structures.

La gestion rigoureuse des erreurs sera valorisée (l'utilisation de `assert` est recommandée), de même que la simplicité, la clarté et la qualité générale du code.

Conformément au programme officiel, il sera admis que les en-têtes standards suivants soient présupposés inclus dans le code : `<assert.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdio.h>`, `<stdlib.h>` et `<string.h>`.

Enfin, dans le contexte du thème étudié, une importance particulière sera accordée à la concision et à l'efficacité du code proposé.

De manière générale, les fragments de code demandés sont courts et excèdent rarement une dizaine de lignes.

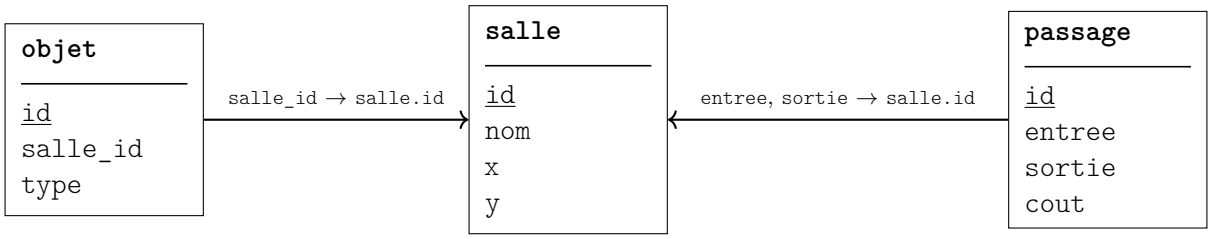
1 Représentation relationnelle des données

Dans un moteur de jeu vidéo, l'une des données centrales est la description de l'espace dans lequel évoluent les entités. On considère ici un ensemble de salles, représentant des zones localisées de l'espace du jeu, reliées entre elles par des passages unidirectionnels formant une structure de navigation. Des objets sont disposés dans les salles, chaque salle pouvant contenir aucun, un ou plusieurs objets.

Afin de décrire de manière précise et cohérente cette structure, les informations correspondantes sont stockées dans une base de données relationnelle, qui permet de représenter les différentes entités manipulées ainsi que les relations qui les lient.

Dans cette partie, on s'intéresse à la représentation relationnelle de ces données et à l'écriture de requêtes SQL simples permettant d'en analyser la structure. Ces requêtes mettront notamment en évidence des relations de voisinage entre salles, qui pourront ensuite être interprétées sous forme de graphe pour les besoins des parties suivantes.

On considère les trois tables relationnelles suivantes :



où :

— `salle.id` est la clé primaire de la table `salle` ;

— `salle.x` et `salle.y` sont des entiers correspondant aux coordonnées du coin supérieur gauche de la salle ;

— `passage.entree` et `passage.sortie` sont des clés étrangères référençant `salle.id` ;

— `passage.cout` est un entier correspondant au coût, pour un joueur, d'emprunter ce passage qui conduit de la salle `passage.entree` vers la salle `passage.sortie` ;

— `objet.salle_id` est une clé étrangère référençant `salle.id` ;

— `objet.type` est une chaîne de caractères décrivant l'objet concerné.

☐ 1 – Expliquer pourquoi la clé étrangère `salle_id` est placée dans `objet` et non l'inverse (c'est-à-dire pourquoi on n'a pas un attribut `objet_id` dans `salle`). Indiquer la cardinalité de l'association correspondante.

☐ 2 – Écrire une requête SQL permettant d'obtenir les noms des salles accessibles directement depuis la salle dont le nom est `'ENTRÉE'`.

☐ 3 – Écrire une requête SQL permettant d'obtenir, pour chaque salle, son nom et le nombre de passages sortants correspondants.

☐ 4 – Écrire une requête SQL permettant d'obtenir, pour chaque salle possédant au moins un passage sortant, son nom et le coût minimal d'un passage sortant depuis cette salle, en triant les résultats par ordre décroissant du coût minimal d'un passage, et en cas d'égalité, par ordre alphabétique sur le nom des salles.

☐ 5 – Écrire une requête SQL permettant d'obtenir, pour chaque salle, son nom et le nombre d'objets présents dans cette salle en ne conservant que les salles contenant au moins deux objets.

2 Représentation en mémoire et navigation dans le monde

L'ensemble des salles et des passages entre les salles peut être modélisé par un graphe orienté pondéré, où les sommets représentent les salles et les arêtes les passages, avec comme pondération le coût de ces derniers. Nous supposerons dans la suite du sujet que les sommets du graphe sont numérotés de `0` à `n-1` pour un graphe à `n` sommets, et que les coûts des arêtes sont strictement positifs.

2.1 Implémentation classique du graphe pondéré

Une représentation classique d'un graphe orienté pondéré en C consiste à utiliser des listes chaînées pour stocker les arêtes sortantes de chaque sommet.

Définition : On considère les structures suivantes :

```
1.  struct _passage {
2.      int cout;
3.      int sortie;
4.      struct _passage* suivant;
5.  };
6.  typedef struct _passage passage;
7.
8.  struct _salle {
9.      int id;
10.     passage* passages;
11. };
12. typedef struct _salle salle;
13.
14. struct _monde {
15.     salle** salles;
16.     int taille;
17. };
18. typedef struct _monde monde;
```

La structure `monde` représente l'ensemble du graphe. Le champ `taille` contient le nombre total de salles du monde. Le champ `salles` est un tableau de pointeurs vers les salles, de taille `taille`.

Chaque sommet du graphe correspond à une salle, représentée par une structure `salle`, qui contient :

— un identifiant unique `id` ;

— un pointeur `passages` vers une liste chaînée de passages sortants.

Chaque passage sortant est représenté par une structure `passage` contenant :

— le coût du passage (`cout`) ;

— l'indice de la salle de destination (`sortie`) ;

— un pointeur `suivant` vers le passage suivant dans la liste.

Ainsi, pour une salle donnée, pointée par un pointeur `s` de type `salle*`, l'ensemble des passages sortants est stocké dans une liste chaînée, parcourue séquentiellement à partir du pointeur `s->passages`. L'absence de passage sortant est représentée par un pointeur `NULL`.

Pour un monde donné, pointé par un pointeur `m` de type `monde*`, on suppose que les salles sont indexées de `0` à `m->taille-1` dans le tableau `m->salles`, et que la valeur `id` d'une salle coïncide avec son indice dans ce tableau, de sorte que `m->salles[id]` est un pointeur vers la salle d'indice `id` du monde.

☐ 6 – Écrire une fonction C de signature

```
monde* monde_init(int n);
```

permettant d'initialiser la structure de données correspondant à un monde comportant `n` salles en allouant dynamiquement la mémoire nécessaire pour le monde et les salles.
On supposera que les listes de passages sont initialement toutes vides.

☐ 7 – Écrire une fonction C de signature

```
int monde_degre(monde* m, int s);
```

qui renvoie le degré sortant de la salle d'indice `s`, c'est-à-dire le nombre de passages sortants issus de cette salle.

☐ 8 – En utilisant la fonction `monde_degre` définie à la question précédente, écrire une fonction C de signature

```
int monde_sommeDegres(monde* m);
```

qui renvoie le nombre total de passages sortants présents dans le monde.

☐ 9 – Écrire une fonction C de signature

```
int monde_coutTotal(monde* m);
```

qui renvoie le coût total du monde, c'est-à-dire la somme des coûts de tous les passages sortants présents dans le monde.

☐ 10 – On considère un monde comportant `ns` salles et `np` passages.
En supposant que :

— chaque entier est encodé sur 32 bits ;

— chaque pointeur est encodé sur 32 bits ;

exprimer en octets l'occupation mémoire totale de cette représentation en fonction de `ns` et `np`.

☐ 11 – On se place dans un contexte très contraint en mémoire, typique des moteurs de jeux vidéo anciens, et l'on suppose que l'on dispose de **64 Ko** de mémoire pour stocker l'ensemble des structures du graphe, représenté à l'aide des listes chaînées décrites précédemment.
On suppose que :

— chaque entier est codé sur 32 bits ;

— chaque pointeur est codé sur 32 bits ;

— en moyenne, chaque salle possède un nombre constant de quatre passages sortants.

Sans effectuer de calcul détaillé, on cherche uniquement à déterminer un **ordre de grandeur** du nombre maximal de salles que l'on peut raisonnablement représenter avec cette structure. Indiquer la valeur la plus plausible parmi les ordres de grandeur suivants :

$$10 \quad 100 \quad 1\,000 \quad 10\,000 \quad 100\,000 \quad 1\,000\,000.$$

2.2 Représentation CSR pour un graphe

Dans cette partie, on introduit une représentation compacte du graphe, fondée sur l'utilisation de tableaux contigus en mémoire, souvent utilisée dans les moteurs de jeux vidéo des années 1990 et connue sous le nom de représentation *CSR* (*Compressed Sparse Row*). Elle est particulièrement bien adaptée aux contraintes étudiées précédemment.

Définition : La représentation CSR utilise trois tableaux :

— un tableau `sorties` contenant, de manière contiguë, les indices des salles de destination de tous les passages du monde ;

— un tableau `indices` indiquant, pour chaque salle `s`, l'indice dans le tableau `sorties` où commence la liste de ses passages sortants ;

— un tableau `couts`, de même taille que `sorties`, tel que `couts[k]` soit le coût du passage allant de la salle d'indice `s` vers la salle d'indice `sorties[k]`, pour tout `k` dans l'intervalle $[\![\texttt{indices[s]}; \texttt{indices[s+1]} - 1]\!]$.

Le tableau `indices` est de taille $ns + 1$, où ns est le nombre total de salles du monde. Pour une salle d'indice `s`, ses passages sortants sont stockés dans les cases `sorties[k]` et `couts[k]` pour

$$\texttt{k} \in [\![\texttt{indices[s]}; \texttt{indices[s+1]} - 1]\!].$$

La dernière case `indices[ns]` joue le rôle de sentinelle et indique la fin de la liste des passages sortants de la dernière salle. En particulier, elle contient la valeur np, qui correspond au nombre total de passages du monde. Cette convention permet de traiter uniformément toutes les salles du monde, sans cas particulier.

☐ 12 – Dessiner le graphe associé à la représentation CSR suivante :

```
sorties = {1,3,2,0,4,1,2},
indices = {0,2,3,5,6,7},
couts   = {2,5,1,4,3,6,2}
```

Définition : On considère la structure suivante :

```
1.  struct _mondeCSR {
2.    int* sorties;
3.    int* indices;
4.    int* couts;
5.    int nb_salles;
6.  };
7.  typedef struct _mondeCSR mondeCSR;
```

☐ 13 – Écrire une fonction C de signature

```
mondeCSR* mondeCSR_init(int ns, int np);
```

qui alloue la mémoire nécessaire pour une structure CSR pour `ns` salles et `np` passages et renvoie un pointeur vers cette structure.
On demande uniquement l'allocation de la mémoire pour la structure et les tableaux internes. Aucune initialisation du contenu n'est attendue à ce stade.

☐ 14 – Écrire une fonction C de signature,

```
int mondeCSR_degre(mondeCSR* mc, int s);
```

permettant de calculer le nombre de passages sortants de la salle d'indice `s`.

☐ 15 – Écrire une fonction C de signature,

```
int mondeCSR_sommeDegres(mondeCSR* mc);
```

qui renvoie la somme des degrés sortants de toutes les salles du monde représenté en CSR, c'est-à-dire le nombre total de passages sortants présents dans le monde.

☐ 16 – Écrire une fonction C de signature,

```
int mondeCSR_coutTotal(mondeCSR* mc);
```

qui renvoie le coût total du monde représenté en CSR, c'est-à-dire la somme des coûts de tous les passages présents dans le monde.

☐ 17 – On souhaite convertir un monde représenté par listes chaînées (structure `monde`) en une représentation CSR (structure `mondeCSR`).
Écrire une fonction C de signature,

```
mondeCSR* monde_vers_CSR(monde* m);
```

qui renvoie un pointeur vers une structure `mondeCSR` représentant le même monde que `m` en copiant l'ensemble des passages et de leurs coûts. On veillera à bien allouer la mémoire pour la nouvelle structure.

☐ 18 – On considère un monde comportant ns salles et np passages, représenté à l'aide de la représentation CSR décrite précédemment.
En supposant que les tableaux `sorties`, `indices` et `couts` contiennent des entiers codés sur 32 bits, exprimer en octets l'occupation mémoire totale de cette représentation en fonction de ns et np.

☐ 19 – Comparer l'occupation mémoire de la représentation du monde par listes chaînées (question 10) et de la représentation CSR (question 18).
On exprimera les deux occupations mémoire en fonction du nombre de salles ns et du nombre de passages np. En tirer une conclusion.

2.3 Recherche d'un chemin de coût minimal

Dans cette sous-partie, on suppose que le graphe représentant le monde est *fortement connexe* : depuis toute salle, on peut atteindre toute autre salle par une suite de passages.

On cherche d'abord à construire rapidement, à partir d'une salle de départ `s`, un chemin *raisonnable* vers chaque salle, sans viser nécessairement le coût minimal.

☐ 20 – Expliquer comment un parcours en largeur du monde, en partant de la salle `s`, permet d'obtenir, pour chaque salle `t`, un chemin utilisant un *nombre minimal de passages* entre `s` et `t`.
Donner un cas dans lequel les chemins sont aussi de *coût minimal*.

Définition : On suppose que l'on dipose d'une structure de file FIFO permettant de stocker des entiers, définie par la structure et les fonctions suivantes :

```
1.  typedef struct _file file;
2.
3.  file* file_init(int taille);
4.  bool  file_vide(file *f);
5.  void  file_enfile(file *f, int x);
6.  int   file_defile(file *f);
7.  void  file_libere(file *f);
```

On peut donc utiliser la structure et ses fonctions librement sans les coder.

- Un appel `file_init(n)` renvoie un pointeur vers une file vide pour laquelle on a alloué de la mémoire nécessaire pour contenir jusqu'à `n` éléments.
- Un appel `file_vide(f)` permet de vérifier si la file pointée par `f` est vide.
- Un appel `file_enfile(f,x)` permet d'ajouter l'entier `x` à la file pointée par `f`.
- Un appel `file_defile(f)` permet de supprimer et renvoyer l'entier de la file pointée par `f` qui a été ajouté en premier.
- Un appel `file_libere(f)` permet de libérer la mémoire allouée pour la file pointée par `f`.

☐ 21 – Écrire une fonction C de signature

```
int* bfs_cout(mondeCSR* mc, int s);
```

qui renvoie un tableau `dist` de taille `mc->nb_salles` tel que `dist[u]` vaut la somme des coûts le long du chemin utilisant *un nombre minimal de passages* trouvé par parcours en largeur entre `s` et `u`. On veillera à allouer la mémoire nécessaire pour ce tableau dynamiquement.
On ne demande pas la reconstruction des chemins, seulement les distances dans le tablau `dist`.

On souhaite maintenant calculer les *coûts minimaux* des chemins depuis une salle source `s`. On va améliorer les valeurs obtenues précédemment par un parcours en largeur en appliquant l'algorithme de Dijkstra.

☐ 22 – On a calculé, à l'aide de la fonction `bfs_cout(mc, s)`, un tableau `dist` tel que, pour toute salle `u`, `dist[u]` soit le coût d'un chemin allant de `s` à `u` comportant un nombre minimal de passages. Expliquer pourquoi, pour toute salle `u`, la valeur `dist[u]` constitue une *borne supérieure* du coût minimal d'un chemin allant de `s` à `u`.

On rappelle que l'algorithme de Dijkstra permet de calculer le coût minimal d'un chemin d'un sommet `s` à un sommet `t` dans un graphe en mettant à jour un tableau de distances `dist` initialement rempli de $+\infty$ et en réalisant un parcours du graphe depuis `s` guidé par une file de priorité jusqu'à atteindre `t`. Un sommet découvert `u` est inséré dans la file de priorité avec une priorité `dist[u]` et les sommets sont explorés dans l'ordre dans lequel ils sont défilés de la file de priorité. Au cours de l'exploration `dist` est mis à jour de sorte qu'en fin d'algorithme `dist[u]` vaut le coût minimal d'un chemin de `s` à `u` pour un sommet `u` plus proche en terme de coût de `s` que `t` et un majorant de ce coût pour un sommet `u` plus loin en terme de coût.

Ici, au lieu de remplir le tableau `dist` de $+\infty$ pour l'initialisation, on utilisera le tableau `dist` calculé par `bfs_cout(mc, s)`. De plus, dans la question suivante, on poursuit l'algorithme sans sommet d'arrivée `t` pour traiter tous les sommets. Le reste de l'algorithme est inchangé.

Définition : On suppose que l'on dipose d'une structure de tas min permettant de stocker des indices de salle associés à une priorité entière, définie par la structure et les fonctions suivantes :

```
1.  typedef struct _tasMin tasMin;
2.
3.  tasMin* tasMin_init(void);
4.  bool    tasMin_vide(tasMin *tm);
5.  void    tasMin_empile(tasMin *tm, int salle, int priorite);
6.  int     tasMin_depile(tasMin *tm);
7.  void    tasMin_libere(tasMin *tm);
```

On peut donc utiliser la structure et ses fonctions librement sans les coder.

- Un appel `tasMin_init()` renvoie un pointeur vers un tas vide.
- Un appel `tasMin_vide(tm)` permet de vérifier si le tas pointé par `tm` est vide.
- Un appel `tasMin_empile(tm,salle,priorite)` permet d'ajouter la salle d'indice `salle` au tas pointé par `tm` avec la priorité `priorite`.
- Un appel `tasMin_depile(tm)` permet de supprimer et renvoyer l'indice de la salle de priorité minimale dans le tas pointé par `tm`.
- Un appel `tasMin_libere(tm)` permet de libérer la mémoire allouée pour le tas pointé par `tm`.

☐ 23 – Écrire une fonction C de signature

```
void dijkstra(mondeCSR* mc, int s, int* dist);
```

qui met à jour le tableau `dist` de sorte que, à l'issue de l'exécution, `dist[u]` contienne le coût minimal d'un chemin allant de la salle d'indice `s` à la salle d'indice `u` pour tout salle. On suppose que le tableau `dist` donné en argument est le tableau calculé par `bfs_cout(mc, s)` et on ne le vérifie pas.
On ne demande pas de reconstruire les chemins.

3 Recherche rapide d'un chemin avec heuristique

Dans un moteur de jeu vidéo, la recherche de chemins doit souvent être effectuée en temps réel. Il est donc crucial de limiter le nombre de salles explorées lors de cette recherche, en particulier lorsque la destination est connue à l'avance.

L'algorithme de Dijkstra, étudié précédemment, explore le monde en fonction du coût accumulé depuis la salle de départ, sans tenir compte de la position de la destination. L'algorithme A* améliore ce comportement en guidant l'exploration à l'aide d'une *heuristique*, c'est-à-dire une estimation du coût restant jusqu'à la salle cible.

Rappels. On rappelle que l'algorithme A* fonctionne de la même façon que l'algorithme de Dijkstra en utilisant en plus une fonction heuristique h qui est définie sur l'ensemble des sommets du graphe. Au lieu de donner une priorité `dist[u]` à un sommet `u`, on lui donne une priorité `dist[u]`$+h($`u`$)$ au moment de l'insérer dans la file de priorité.

Notations. Dans toute cette partie, on considère :

- une salle source d'indice `s` et une salle destination d'indice `t` ;
- pour chaque salle d'indice `u`, une valeur `dist[u]` représentant le coût actuellement connu d'un chemin allant de la salle d'indice `s` à la salle d'indice `u` ;
- une heuristique `h(u)` estimant le coût restant entre une salle d'indice `u` et la salle destination d'indice `t`. On suppose que `h(t) = 0` ;
- $\delta($`u`$,$ `t`$)$ le coût minimal réel d'un chemin allant d'une salle d'indice `u` à la salle d'indice `t`.

On s'intéresse d'abord aux propriétés des heuristiques utilisées par l'algorithme A*, et à leurs conséquences sur la correction et l'efficacité de l'algorithme.

Définition : Une heuristique `h` est dite *admissible* si, pour toute salle `u`, la valeur `h(u)` est inférieure ou égale au coût du meilleur chemin possible allant de `u` vers la salle destination `t`. Autrement dit, pour toute salle `u` :

$$\texttt{h(u)} \leqslant \delta(\texttt{u}, \texttt{t}).$$

☐ 24 – On suppose que l'algorithme A* utilise une heuristique admissible.
En admettant la terminaison de l'algorithme A* et le fait qu'un chemin peut être reconstruit à partir du tableau, expliquer pourquoi le chemin trouvé est nécessairement de coût minimal dès lors que la salle destination `t` est extraite de la file de priorité.
On pourra notamment s'appuyer sur :

— le fait que les salles sont extraites selon la valeur `dist[u] + h(u)` ;

— le rôle de l'admissibilité de l'heuristique ;

— un raisonnement par l'absurde.

☐ 25 – Préciser un lien entre l'algorithme de Dijkstra et l'algorithme A* qui permet d'expliquer en quoi ce raisonnement permet également de justifier l'optimalité du chemin obtenu avec l'algorithme de Dijkstra.

Définition : Une heuristique `h` est dite *monotone* si `h(t)` $= 0$ et pour tout passage d'une salle d'indice `u` à une salle d'indice `v` de coût `c`, on a :

$$\texttt{h(u)} \leqslant \texttt{c} + \texttt{h(v)}.$$

☐ 26 – Montrer que toute heuristique monotone est nécessairement admissible
On attend un raisonnement exploitant le suivi d'un chemin dans le graphe, faisant apparaître une chaîne d'inégalités reliant `h(u)` au coût total du chemin.

☐ 27 – Expliquer l'intérêt pratique de la monotonie d'une heuristique dans l'algorithme A*.
En particulier, préciser en quoi la monotonie permet :

— de garantir qu'une salle extraite de la file de priorité possède un coût `dist` définitif ;

— d'éviter de ré-insérer des salles déjà traitées.

3.1 Heuristique pondérée : compromis entre rapidité et qualité

Afin d'accélérer la recherche, on considère une variante de l'algorithme A*, appelée *A* pondéré* (*Weighted A**), dans laquelle l'heuristique est multipliée par un facteur $w > 1$.

Dans cette variante, la priorité associée à une salle `u` lors de son insertion dans le tas est donnée par :

$$f(\texttt{u}) = \texttt{dist[u]} + w \cdot \texttt{h(u)}.$$

Pour comprendre l'intérêt de cette variante, il est nécessaire de revenir sur le rôle de l'heuristique dans l'algorithme A* classique.

☐ 28 – Comparer l'ordre d'exploration des salles dans l'algorithme de Dijkstra et dans l'algorithme A*.
On précisera :

— selon quelle valeur les salles sont extraites dans chaque algorithme ;

— en quoi l'introduction de l'heuristique modifie l'exploration du graphe par rapport à la salle destination.

☐ 29 – Dans l'algorithme A* pondéré, la priorité est donnée par :

$$f(\texttt{u}) = \texttt{dist[u]} + w \cdot \texttt{h(u)}, \quad \text{avec } w > 1.$$

Expliquer en quoi l'introduction du facteur w modifie le poids relatif de l'heuristique par rapport au coût déjà parcouru, et quel effet cela a sur l'ordre d'exploration des salles.

☐ 30 – On considère une heuristique admissible `h`, et un facteur $w > 1$. On définit une nouvelle heuristique pondérée par :

$$h_w(\texttt{u}) = w \cdot \texttt{h(u)}.$$

Indiquer si on peut toujours affirmer que l'heuristique h_w est encore admissible lorsque $w > 1$, et justifier brièvement la réponse.

☐ 31 – Expliquer l'effet de l'augmentation du paramètre w sur le comportement de l'algorithme, et en particulier sur :

— le nombre de salles explorées ;

— la qualité du chemin trouvé. Est-il toujours optimal ?

☐ 32 – En vous appuyant sur les questions précédentes expliquer pourquoi cette variante est particulièrement adaptée aux moteurs de jeux vidéo anciens, cherchant un compromis entre rapidité de calcul et qualité du chemin.

Définition : On dit qu'un chemin allant de la salle source d'indice `s` à la salle destination d'indice `t` et de coût c est *w-optimal* (avec $w > 1$) si

$$c \leqslant w \cdot \delta(\texttt{s}, \texttt{t}),$$

où $\delta(\texttt{s}, \texttt{t})$ désigne le coût minimal réel d'un chemin allant de la salle d'indice `s` à la salle d'indice `t`.

Cette notion introduit une tolérance contrôlée par le paramètre w sur la qualité du chemin retourné par l'algorithme.

On considère maintenant un algorithme A* pondéré avec un facteur $w > 1$. Les salles sont extraites de la file de priorité selon la valeur

$$f(\texttt{u}) = \texttt{dist[u]} + w \cdot \texttt{h(u)},$$

où `dist[u]` est le coût du chemin courant de la salle d'indice `s` à la salle d'indice `u`, et `h(u)` une heuristique admissible estimant le coût restant jusqu'à la salle d'indice `t` depuis la salle d'indice `u`.

L'algorithme maintient une variable `best`, qui contient à tout instant le coût du meilleur chemin complet trouvé jusqu'à présent entre la salle source d'indice `s` et la salle destination `t`. Cette variable est mise à jour chaque fois qu'un nouveau chemin complet vers la salle d'indice `t` de coût plus faible est découvert.

On considère l'extrait du code de l'algorithme suivant :

```
1.  /* Extrait de l'algorithme */
2.  int u = tas_depile(t);
3.
4.  /* Condition : arrêt anticipé */
5.  if (dist[u] + w * h(u) >= best) {
6.      break;
7.  }
```

☐ 33 – Montrer que l'arrêt de l'algorithme lorsque la condition précédente est satisfaite garantit que le chemin retourné, de coût `best`, est *w-optimal*.
On pourra procéder en deux étapes :

— considérer un chemin optimal reliant la salle source d'indice `s` à la salle destination d'indice `t`, et montrer que, pour toute salle d'indice `v` située sur ce chemin, la valeur $f(\mathtt{v})$ est majorée par $w \cdot \delta(\mathtt{s}, \mathtt{t})$;

— en déduire que, tant qu'une telle salle d'indice `v` existe dans la file de priorité, la priorité minimale vérifie $\min f(\cdot) \leqslant w \cdot \delta(\mathtt{s}, \mathtt{t})$, puis expliquer comment la condition d'arrêt

$$\min f(\cdot) \geqslant \mathtt{best}$$

permet de conclure que `best` vérifie la définition d'un chemin *w-optimal*.

4 Arbres BSP et dérécursivation

Dans les moteurs de jeux vidéo des années 1990, les structures arborescentes étaient largement utilisées pour organiser l'espace et accélérer les calculs de visibilité, de collisions ou de rendu. Les arbres BSP (*Binary Space Partitioning*) constituent un exemple emblématique de telles structures.

Définition : Un arbre BSP (*Binary Space Partitioning*) est une structure arborescente permettant de partitionner récursivement l'espace à l'aide de séparations géométriques (droites en 2D, plans en 3D). Chaque nœud interne représente une séparation de l'espace en deux régions disjointes, et les feuilles correspondent à des régions élémentaires contenant des objets ou des primitives géométriques.

4.1 Dérécursivation

Dans cette partie, on s'intéresse au parcours d'un arbre BSP, ainsi qu'à la transformation d'algorithmes récursifs en versions itératives par déroulage explicite de la récursivité, afin de mieux contrôler l'utilisation de la mémoire.

On représente un arbre BSP par un tableau `bsp` de nœuds. Chaque nœud est identifié par son indice `n` et possède deux champs :

— `filsDevant` : indice du fils gauche ou fils « devant », ou `-1` s'il n'existe pas ;

— `filsDerriere` : indice du fils droit ou fils « derrière », ou `-1` s'il n'existe pas.

On se donne la structure suivante en C :

```
1.  struct _noeud{
2.      int filsDevant;
3.      int filsDerriere;
4.  };
5.  typedef struct _noeud noeud;
```

On suppose disponible une fonction

```
void visite(int n);
```

qui effectue le traitement associé à un nœud d'indice `n` dans un arbre BSP.

On suppose ici qu'un arbre BSP est accessible via une variable globale `bsp`, de type tableau de nœuds et on souhaite écrire une fonction `infixe`, de signature

```
void infixe(int n);
```

permettant de parcourir l'arbre BSP selon un parcours infixe.

On rappelle qu'un parcours infixe consiste, pour un nœud `n`, à :

1. parcourir le sous-arbre « devant » ;
2. visiter le nœud `n` ;
3. parcourir le sous-arbre « derrière ».

4.1.1 Parcours infixe récursif

☐ 34 – Écrire la version récursive de la fonction `infixe`.

4.1.2 Dérécursivation par déroulage explicite

On souhaite obtenir une version itérative équivalente au parcours infixe précédent, en déroulant explicitement la récursivité.

On utilise une pile explicite implémentée par :

— un tableau d'entiers `pile` ;

— un entier `sommet`, représentant le nombre d'éléments actuellement empilés et tel que le sommet de la pile est en position `sommet-1`.

☐ 35 – On considère le squelette de code suivant, dans lequel l'initialisation de la pile n'est pas précisée :

```
1.  void infixe(int n) {
2.    /* pile : tableau d'indices, sommet : taille courante */
3.
4.    while (true) {
5.      while (bsp[n].filsDevant != -1) {
6.        pile[sommet] = n;
7.        sommet++;
8.        n = bsp[n].filsDevant;
9.      }
10.
11.     while (true) {
12.       visite(n);
13.       if (bsp[n].filsDerriere != -1) {
14.         n = bsp[n].filsDerriere;
15.         break;
16.       }
17.
18.       if (sommet == 0) { return; }
19.
20.       /* L1 */
21.       /* L2 */
22.     }
23.   }
24. }
```

Compléter les lignes **L1** et **L2** de manière à obtenir un parcours itératif équivalent au parcours infixe récursif pour l'ordre de visite des nœuds.

4.2 Génération procédurale d'une carte de jeu

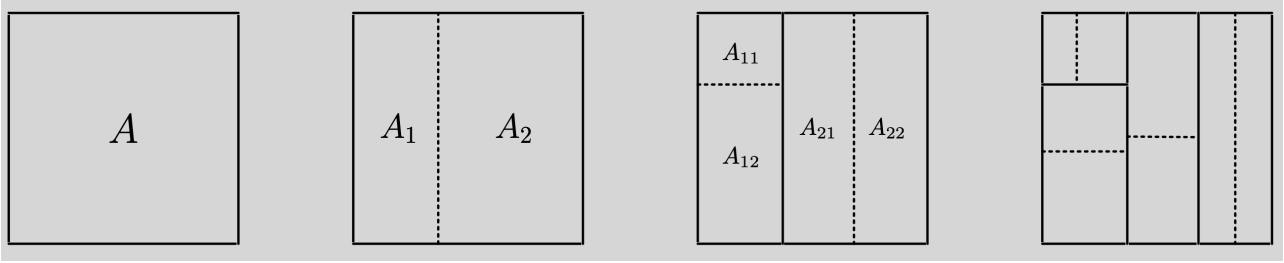
On souhaite à présent utiliser un arbre BSP pour générer procéduralement la structure d'une carte de jeu.

On part d'une salle initiale carrée, représentant l'intégralité de l'espace de jeu. Cette salle est découpée récursivement en deux sous-salles selon :

— une direction choisie aléatoirement (horizontale ou verticale) ;

— une position de coupe choisie aléatoirement entre 30% et 70% de la dimension correspondante.

Ce procédé est répété récursivement sur les sous-salles obtenues, jusqu'à atteindre un nombre de salles égal à une puissance de deux voulue.



Cette étape de génération peut être modélisée par un arbre BSP. Chaque nœud de l'arbre correspond à une opération de découpe de l'espace, et est caractérisé par :

— une direction de coupe représentée par 0 si elle est verticale et 1 si elle est horizontale ;

— une proportion de coupe, stockée sous forme d'un entier compris entre 300 et 700, représentant un pourcentage en ‰.

Les feuilles de l'arbre correspondent aux salles finales obtenues à l'issue du processus.

Dans la figure ci-dessus :

— la racine de l'arbre représente la première découpe, séparant la salle initiale A en deux sous-salles A_1 et A_2 ;

— le fils gauche de la racine correspond à la découpe de A_1 en A_{11} et A_{12} ;

— le fils droit correspond à la découpe de A_2 en A_{21} et A_{22} ;

— et ainsi de suite pour les niveaux suivants de l'arbre.

☐ 36 – On rappelle que la fonction `rand()` renvoie un entier tiré aléatoirement entre `0` et `RAND_MAX`, qui est un entier très grand assimilable ici à $+\infty$, selon une loi uniforme.
Écrire une fonction `alea` ayant pour signature,

```
int alea(void);
```

et renvoyant un entier tiré aléatoirement entre 300 et 700 inclus selon une loi uniforme.

Pour représenter l'arbre BSP en mémoire, on utilise un tableau de `n` nœuds. Les nœuds sont numérotés à partir de `0` pour la racine, puis par niveaux, de gauche à droite et de haut en bas de manière à apparaître dans le tableau dans l'ordre du parcours en largeur de l'arbre BSP. Chaque nœud contient deux entiers :

— un entier valant `0` ou `1`, indiquant la direction de la coupe (`0` : verticale, `1` : horizontale) ;

— un entier compris entre `300` et `700`, représentant la proportion de la coupe en ‰.

Pour la figure précédente, on obtient le tableau suivant :

0	1	0	0	1	1	0
370	690	492	405	580	465	500

☐ 37 – On se donne la structure en C suivante pour les coupes :

```
1.  struct _coupe {
2.    int dir; // 0 ou 1
3.    int proportion; // entre 300 et 700 inclus
4.  };
5.  typedef struct _coupe coupe;
```

Écrire une fonction C de signature ,

```
coupe* repartition(int n);
```

qui initialise un tableau de coupes représentant un arbre BSP de `n` nœuds comme décrit dans cette sous-partie.
Les directions de coupe ainsi que les proportions de coupe sont tirées aléatoirement.
On veillera à allouer dynamiquement la mémoire nécessaire.

On souhaite à présent associer à chaque salle une position géométrique, afin de pouvoir tracer la carte générée.

On se place dans un repère orthonormé. La salle initiale occupe le carré de coordonnées $(0, 0)$ et $(100n, 100n)$, où $n = \log_2(p)$ et p est le nombre final de salles souhaité.

Chaque salle est repérée par un quadruplet

$$(x_{\min}, x_{\max}, y_{\min}, y_{\max}),$$

correspondant aux coordonnées de son coin inférieur gauche $(x_{\min}, y_{\min})$ et de son coin supérieur droit $(x_{\max}, y_{\max})$.

On se donne la structure en C suivant pour définir les région :

```
1.  struct _region {
2.      int xmin;
3.      int xmax;
4.      int ymin;
5.      int ymax;
6.  };
7.  typedef struct _region region;
```

☐ 38 – En utilisant la fonction `repartition` et un arbre BSP binaire complet, écrire une fonction de signature,

```
region* creation(int n);
```

qui renvoie un tableau de `region` contenant les coordonnées de toutes les régions engendrées par un arbre BSP à `n` nœuds généré avec la fonction `repartition`, y compris la région initiale et les régions intermédiaires. On admet que la valeur de `n` est le nombre de nœuds d'un arbre binaire complet et on ne le vérifie pas.
Dans l'exemple illustré précédemment, le tableau retourné contiendra les régions $A, A_1, A_2, A_{11}, A_{12}, A_{21}, A_{22}, \ldots$, soit un total de 15 régions pour 7 coupes donc 7 nœuds.

On peut transformer la partition spatiale obtenue à l'aide de l'arbre BSP en une *carte de jeu exploitable*, composée de salles reliées par des couloirs.

Le principe de génération est le suivant :

1. Dans chaque région élémentaire correspondant à une feuille de l'arbre BSP, on place aléatoirement une salle, contenue strictement dans cette région.
2. Pour chaque nœud interne de l'arbre BSP, on génère un couloir traversant la ligne de séparation correspondante, de manière à assurer la connectivité entre les deux sous-régions associées.

Un couloir peut ainsi relier :

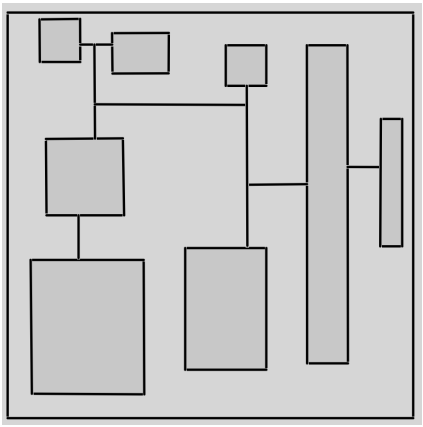
— deux salles situées dans des régions adjacentes ;

— une salle et un couloir déjà généré ;

— deux couloirs issus de niveaux différents de l'arbre BSP.

Cette construction garantit que la carte finale est connexe, tout en conservant une structure hiérarchique héritée de l'arbre BSP.

Un exemple de carte générée selon ce principe est illustré ci-dessous :



Fin de l'épreuve

Travailler les chapitres de ce sujet

Prepa Help suit le programme de MPI chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Langage C : types, tableaux, pointeurs, allocation
- Architecture de l'ordinateur et représentation des nombres
- Récursivité

Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière : **écrire sur WhatsApp**

Les autres années et épreuves : prepahelp.com/annales/mines/mpi