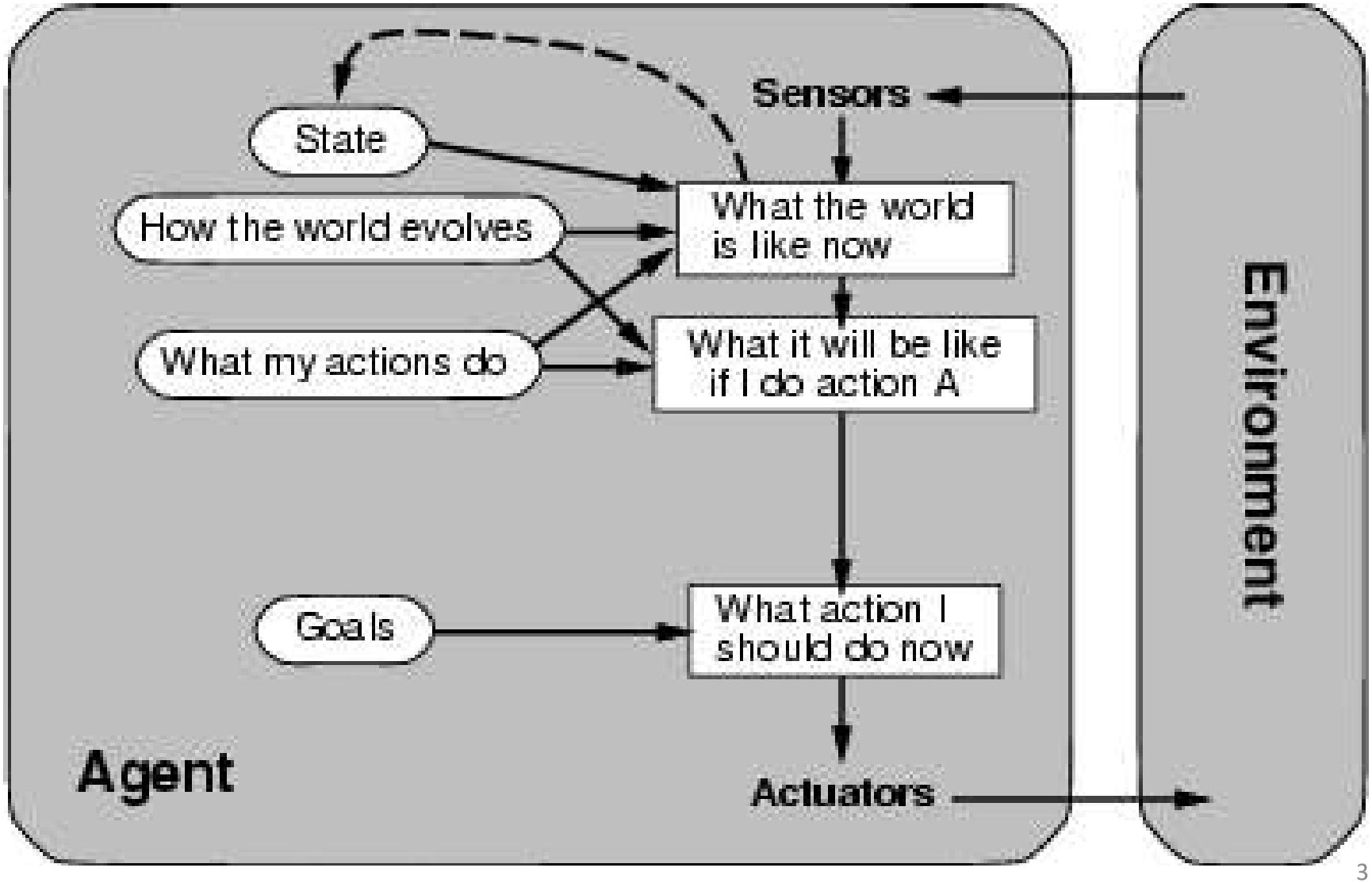
Intelligence Artificielle

Résolution de problèmes par recherche

Comment les humains prennent-ils des décisions ?

1. Observer la situation actuelle.
2. Énumérer les options possibles.
3. Évaluer les conséquences des options (simulation).
4. Retenir la meilleure option possible satisfaisant le but.

Rappel : Agent basé sur des buts



Agent basé buts / Boucle de contrôle

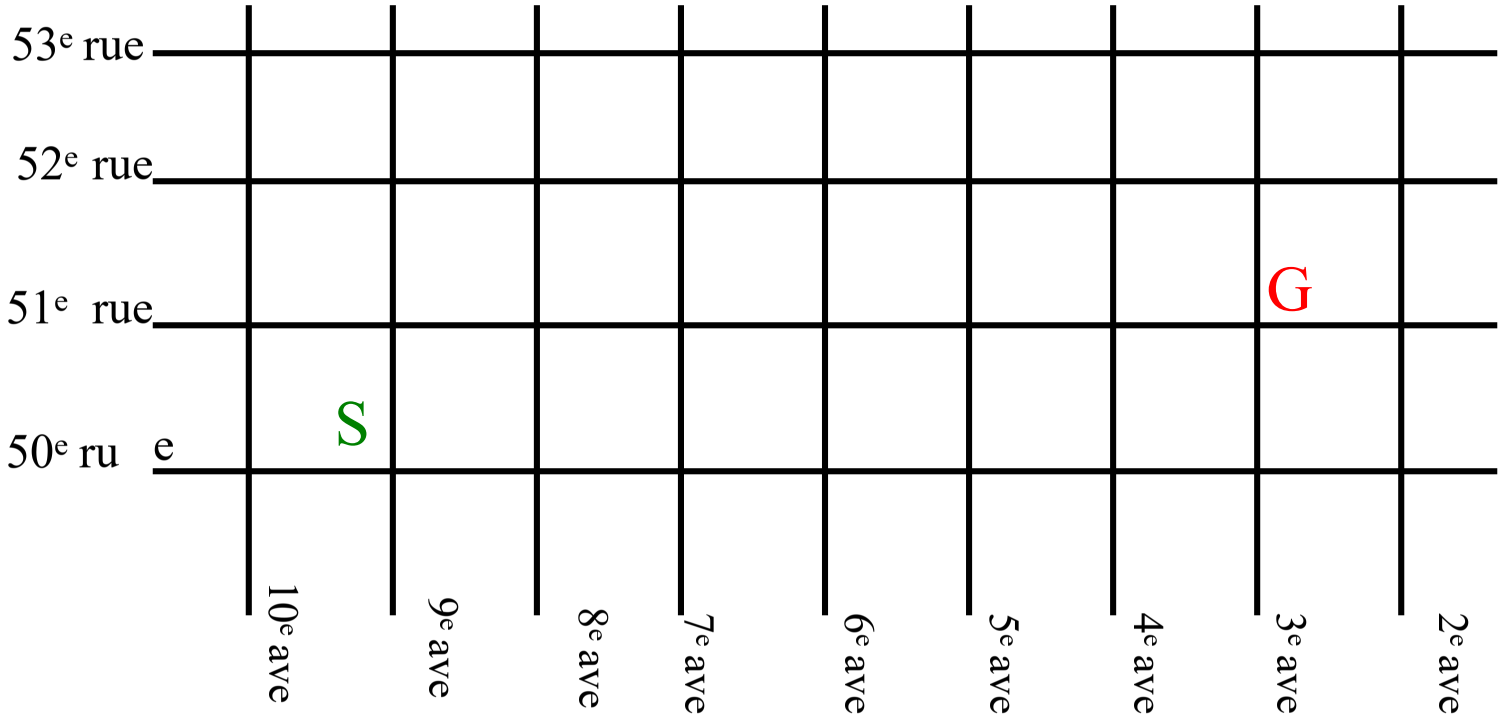
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Application 1: trouver chemin dans ville

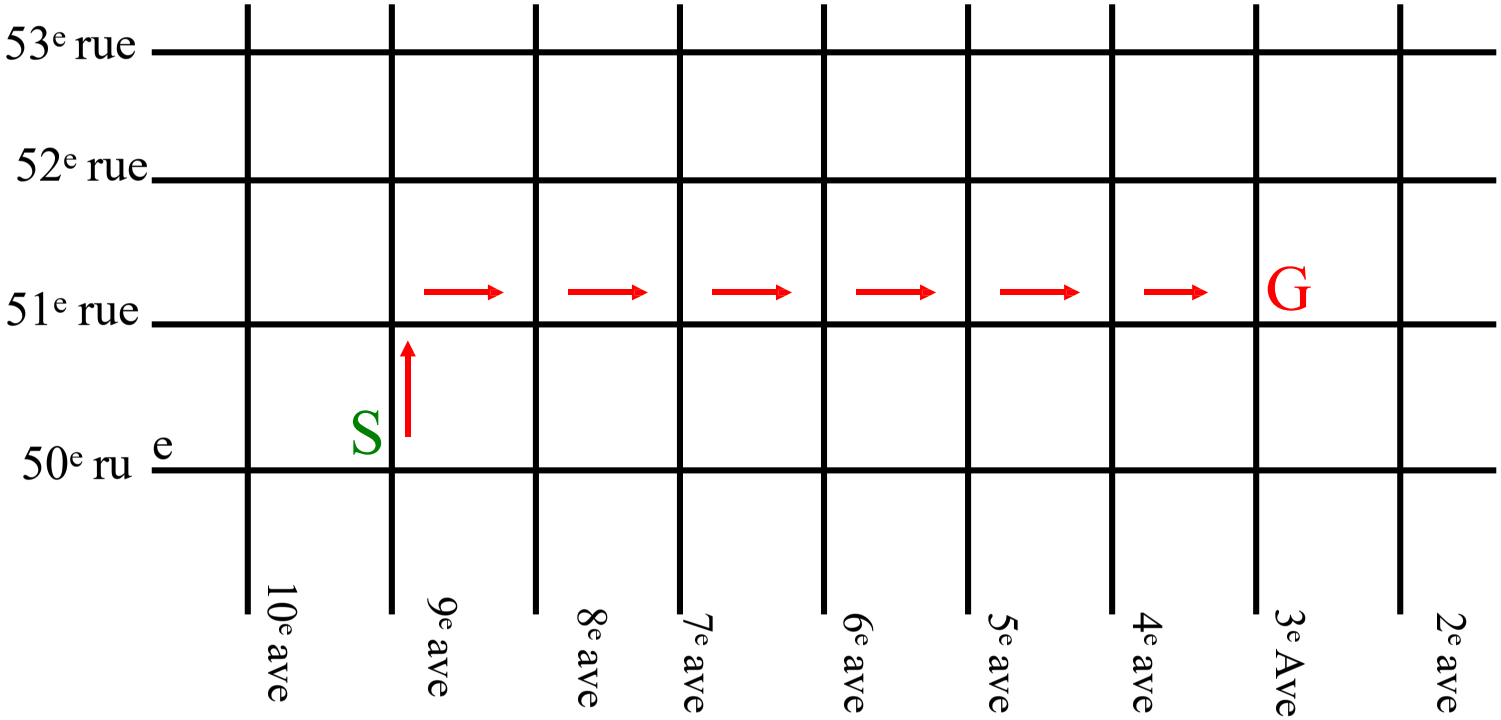
Trouver un chemin de la 9e ave & 50e rue à la 3e ave et 51e rue

Voir schéma

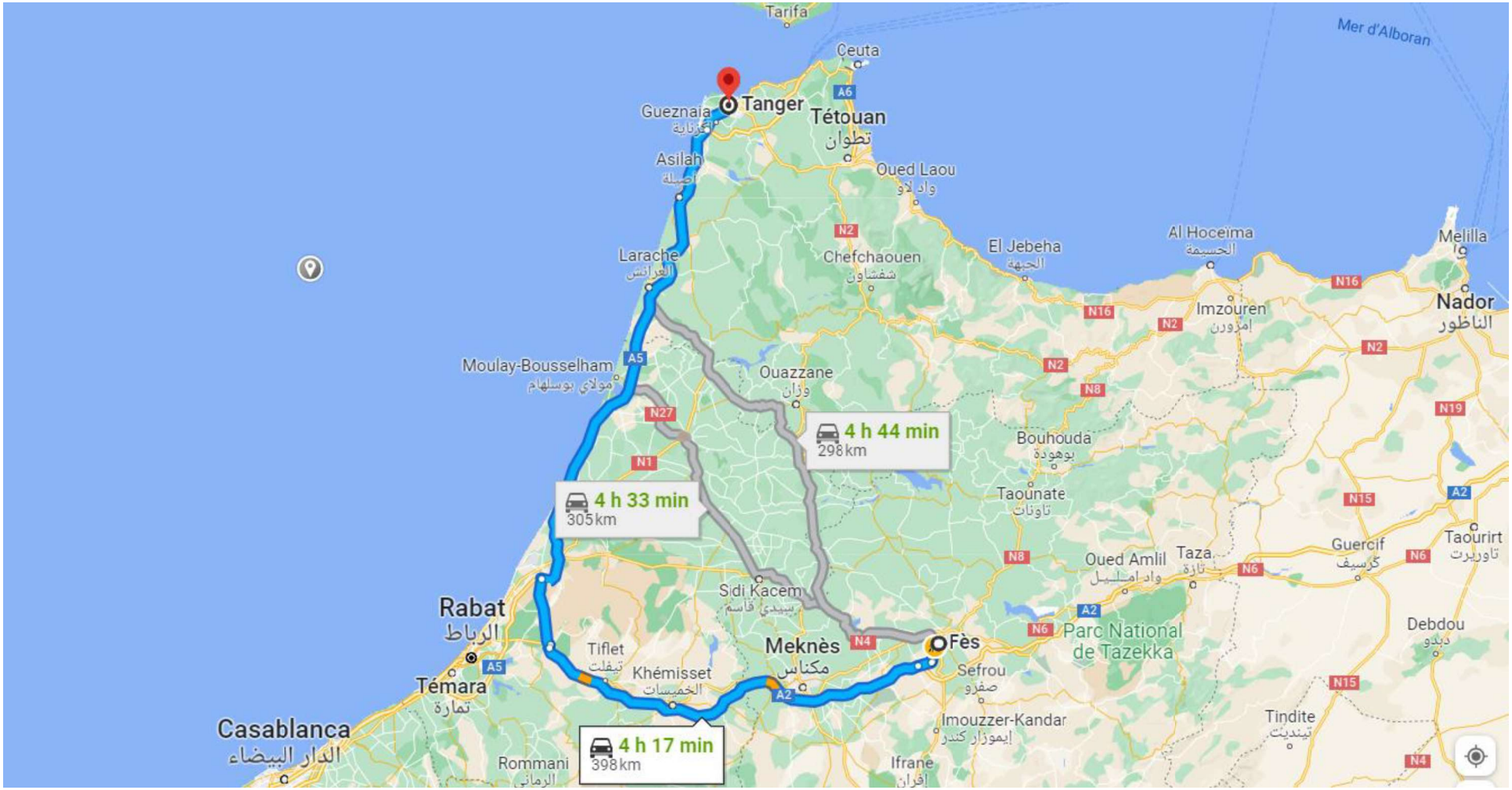


Application 1 – Ville quadrillée

- Nœuds = intersections.
- Arêtes = segments de rue.



Application 2 – Recherche dans une carte

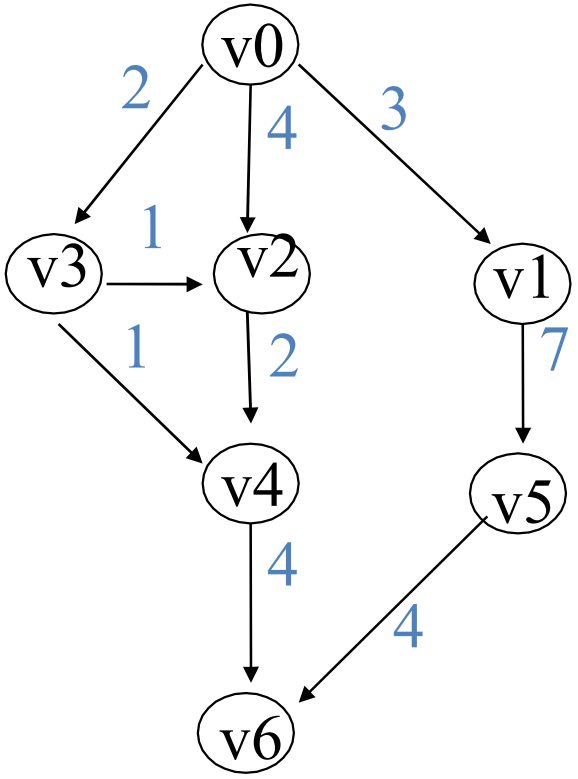


Application 2 – Recherche dans une carte

Domaine :

Routes entre les villes

transitions(v0):
((2,v3), (4,v2), (3, v1))



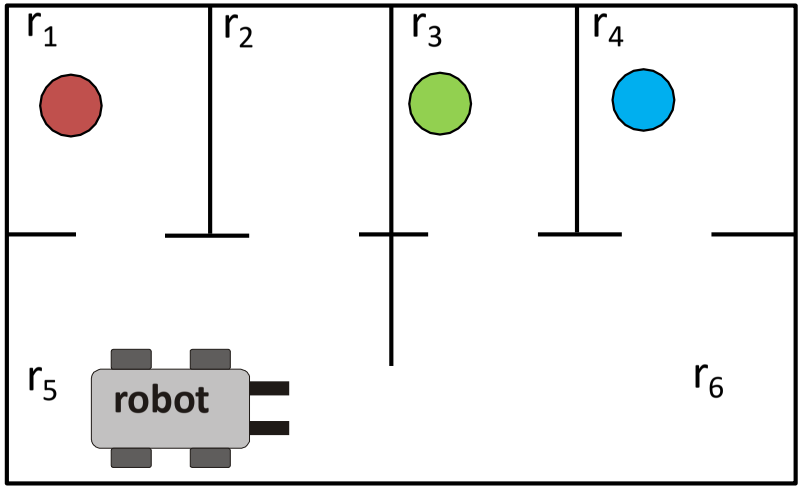
Problème posé (initNode, goal):

v0: ville de départ (état initial)
v6: destination (but)

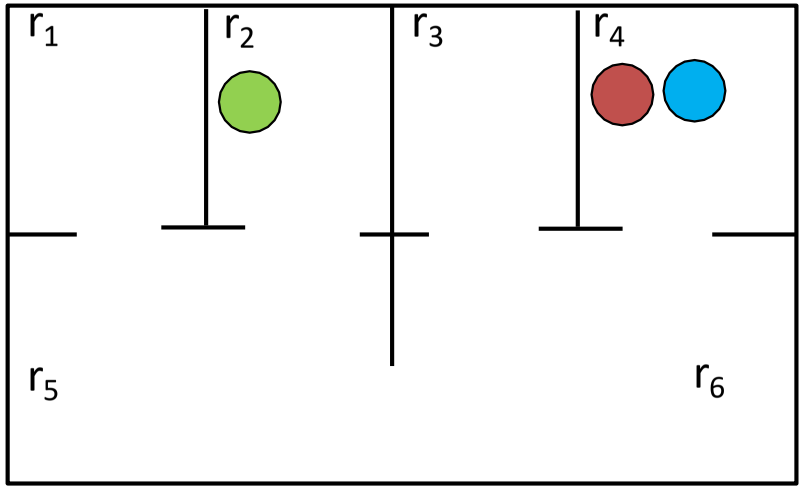
En d'autres termes:
goal(v): vrai si v=v6

Application 3 – Robot-livreur de colis

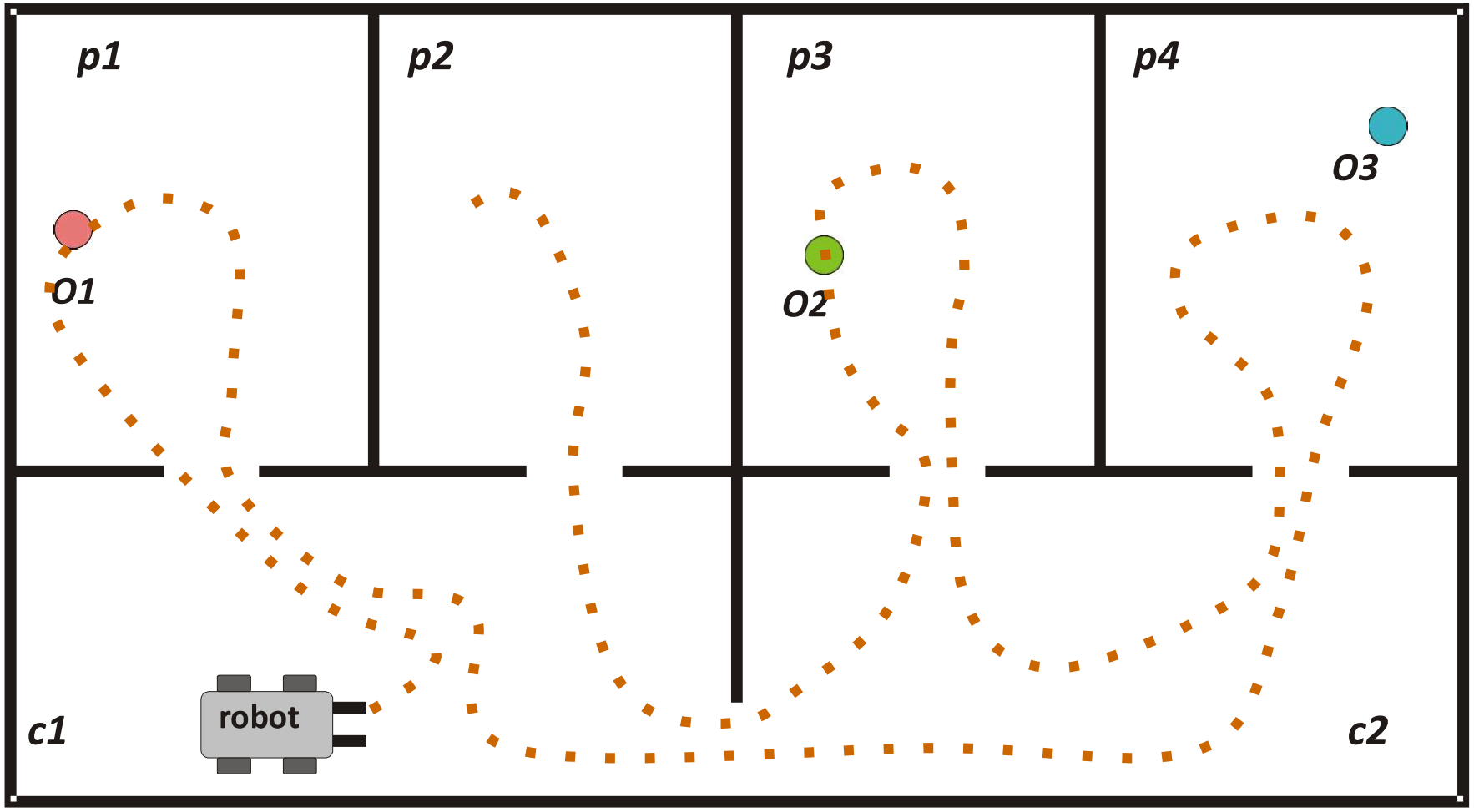
État initial



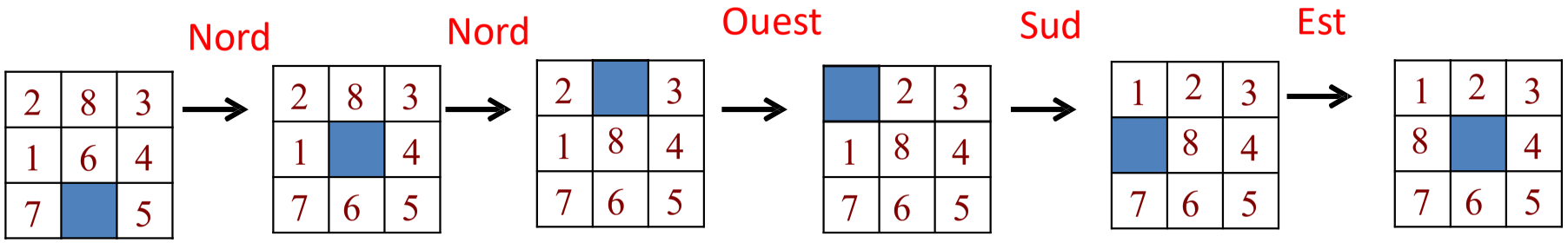
But



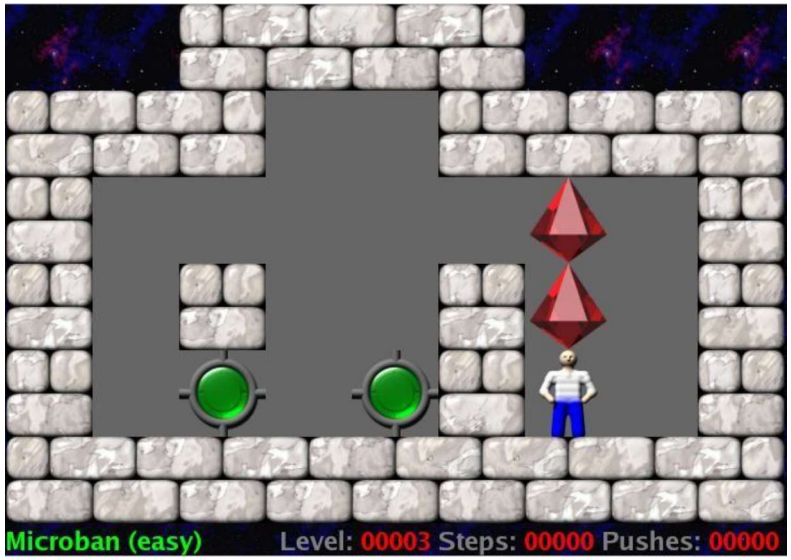
Application 3 – Robot-livreur de colis



Application 4 – Jeu de taquin



Application 5 – Jeu Sokoban



Formalisation d'un problème de recherche

- Entrées
  - Un **nœud (état) initial n_0** (situation initiale)**.**
  - Une fonction **but(*n*)** qui retourne *true* ssi le but est atteint dans le nœud *n*.
  - Une fonction de transition **successeurs(*n*)** qui retourne/génère les nœuds successeurs de *n*.
- Sortie
  - **Solution = chemin dans un graphe** (séquence nœuds / actions)
- Le **coût d'une solution** est la **somme des coûts des arêtes** dans le graphe.
- Il peut y avoir plusieurs nœuds (états) satisfaisant le but.

Critères d'évaluation des algorithmes (stratégies) de recherche

- **Correcte**. La solution retournée est correcte.
- **Complétude**. L'algorithme trouve une solution lorsqu'il en existe une et indique qu'il n'y en a pas lorsque aucune n'existe.
- **Optimalité**. Garantie que la solution retournée est optimale. Alternativement, on peut avoir des garanties d'approximation (solution proche optimale).
- **Complexité temporelle**.
- **Complexité spatiale**.

Algorithmes (stratégies) de recherche

- Recherche non informée :
  - Recherche en **largeur** (Breadth-First-Search)
  - Recherche en **profondeur** (Depth-First-Search)
    - Sans limite de profondeur (version classique)
    - Avec limite de profondeur
    - Incrémentale (Iterative Depth-First-Search)
  - Algorithme de **Dijkstra**
- Recherche informée
  - Sujet suivant.

Recherche en largeur

RechercheLargeur(n_0)

1. Créer file *open*
2. Enfiler n_0 dans *open*
3. Tant que *open* n'est pas vide :
4. n ← défiler *open*
5. marquer n comme visité
6. S ← sucesseurs(n)
7. Pour tout s dans S :
8. Si s n'est pas visité alors :
9. ajouter s dans *open*

Recherche en largeur

- Correcte? **Oui**.
- Complétude. **Oui**.
- Optimalité. **Oui,** si les coûts sont uniformes. Si non uniforme, requiert d'avoir une file prioritaire.
- Complexité temporelle. **$O(b^d)$** où b est le facteur de branchement et d la profondeur de la solution.
- Complexité spatiale. **$O(b^d)$**

Recherche en profondeur (Arbre)

RechercheProfondeur(n)

S ← sucesseurs(n)

3. Pour tout n’ dans S :

4. Si n’ n’est pas sur la pile de visite:

5. RechercheProfondeur(n’)

Recherche en profondeur (Arbre)

- Correcte? **Oui**.
- Complétude. **Oui**, si espace d'états finis et détection les boucles.
- Optimalité. **Non**.
- Complexité temporelle. **$O(b^m)$** où *b* est le facteur de branchement et *m* la profondeur maximale. Généralement : m>d.
- Complexité spatiale. **O(bm)** pour détection de boucles. Il y a moyen de réduire à **O(b)** en itérant sur les successeur sans tous les générer.

Recherche en profondeur (Graphe)

RechercheProfondeur(n)

1. Marquer n visité (ou ajouter n à l'ensemble des nœuds explorés)
2. S ← sucesseurs(n)
3. Pour tout n' dans S :
4. Si n' n'est pas visité alors :
5. RechercheProfondeur(n')

Recherche en profondeur (Graphe)

- Correcte? **Oui**.
- Complétude. **Oui**.
- Optimalité. **Non**.
- Complexité temporelle. **O(|S|)** où |S| est la taille de l'espace d'états. Remarque:$|S| \leq b^d$.
- Complexité spatiale. **O(|S|).**

Recherche en profondeur avec profondeur limitée (Arbre/Graphe) et coûts uniformes

```
RechercheProfondeurLimité (n):
1.    for i = 1 .. k     (garantie complet si k=|S|)
2.        solution ← RechercheProfondeur(n, i)
3.        if(solution ≠ échec)
4.              retourner solution

RechercheProfondeur(n, maxprof)
1.  si maxprof < 0 return
2.  marquer n visité
3.   S ← sucesseurs(n)
4.      Pour tout s dans S :
5.            si s n'est pas visité alors :
6.                 RechercheProfondeur(n,maxprof-1)
```

Recherche en profondeur itérative (graphe)

- Correcte? **Oui**.
- Complétude. **Oui**, si poussé jusqu'à k=|S|
- Optimalité. **Oui,** si les coûts sont uniformes. Si non uniforme, requiert quelques modifications.
- Complexité temporelle. **O(|S|)** où |S| est la taille de l'espace d'états. Remarque:$|S| \leq b^d$.
- Complexité spatiale. **O(|S|).** En pratique, nécessite beaucoup moins de mémoire que DFS.