Inteligência Artificial para Robótica Móvel

Busca Informada

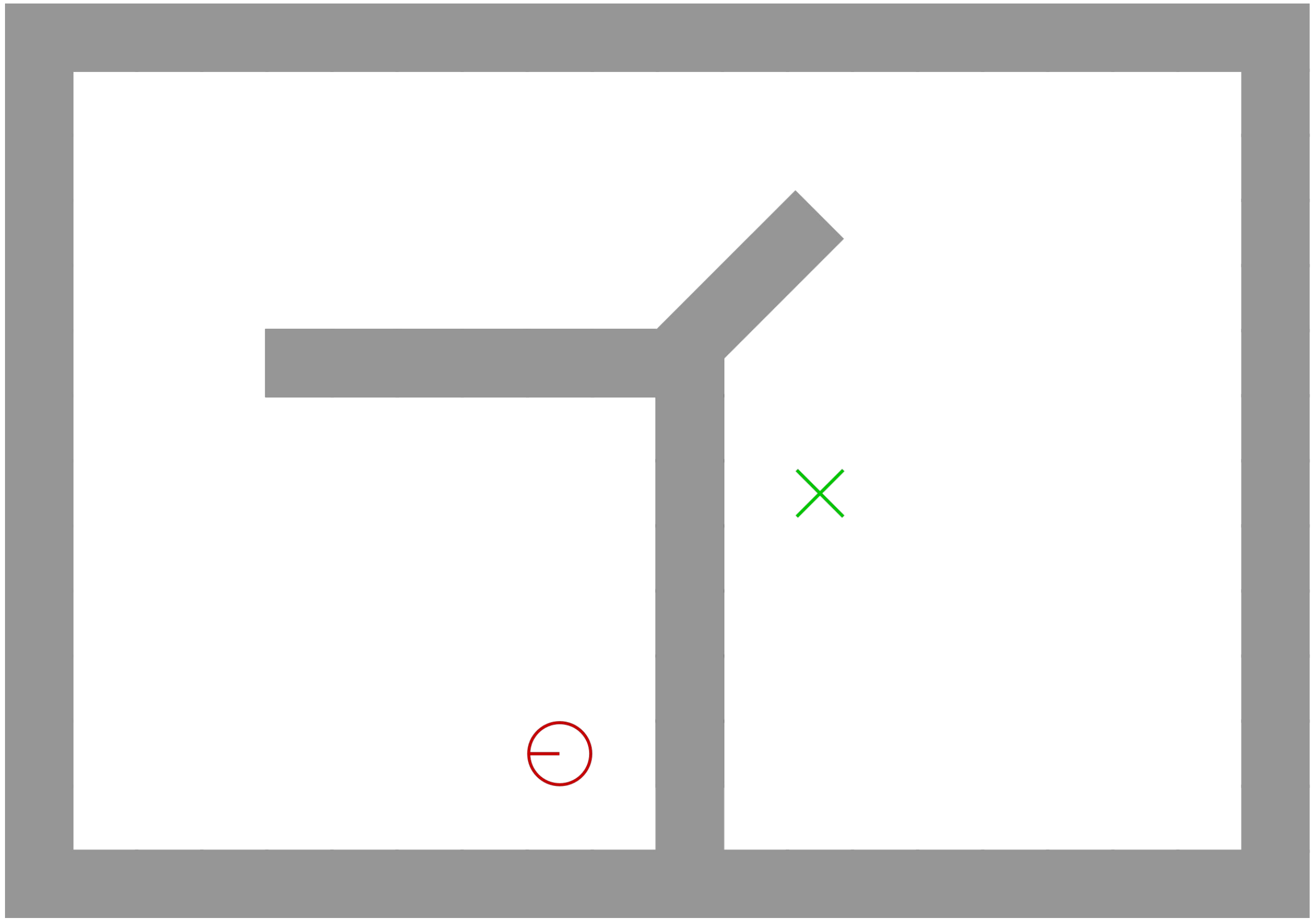
Professor: Marcos Maximo

Roteiro

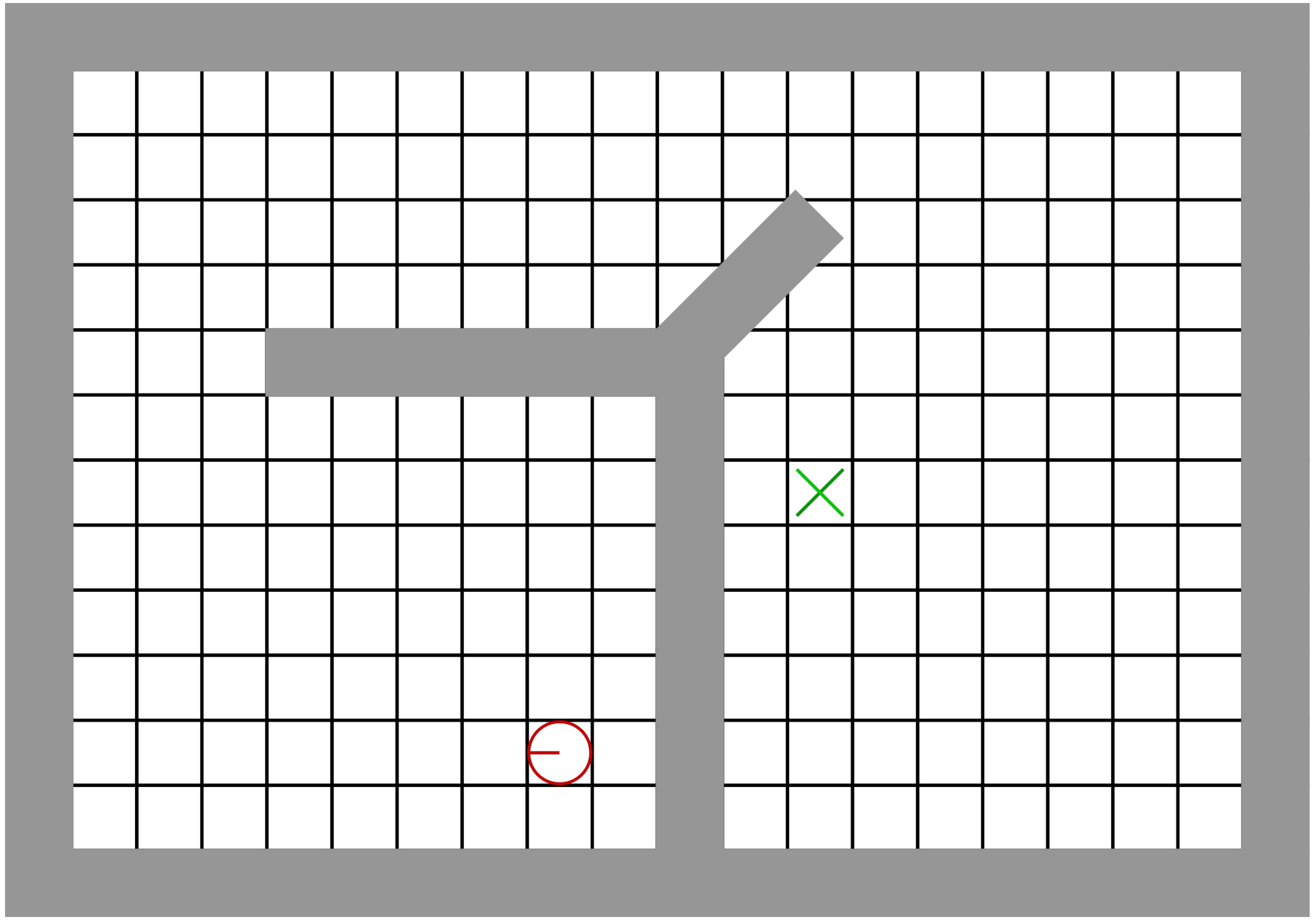
- Motivação.
- Busca em Árvore.
- Revisão de grafos.
- Busca em Grafos.
- Algoritmo de Dijkstra.
- Busca Informada.
- Geração de Grafos para Planejamento de Caminho.
- Planejamento de Ações no Futebol de Robôs.

Motivação

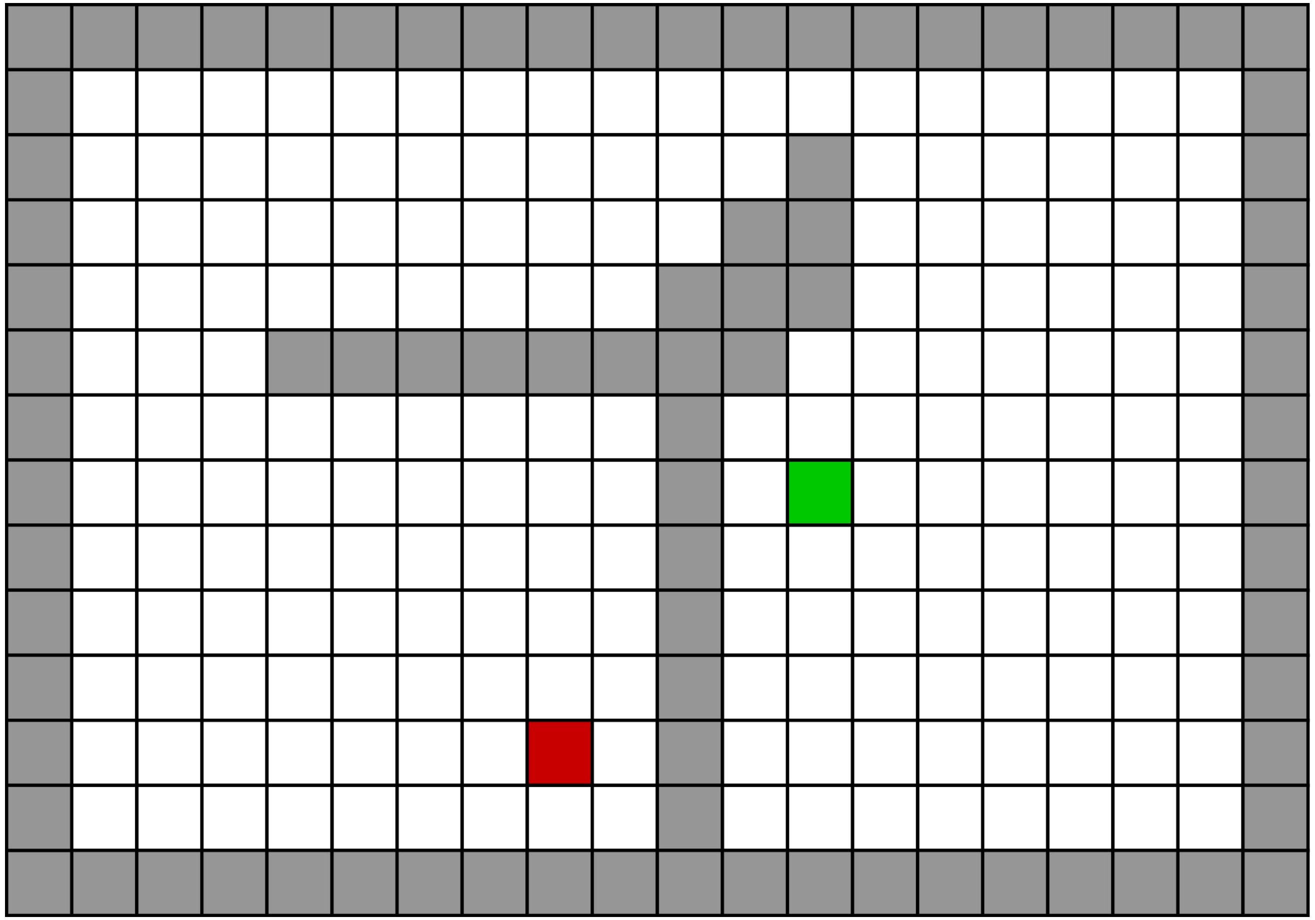
Planejamento de Caminho



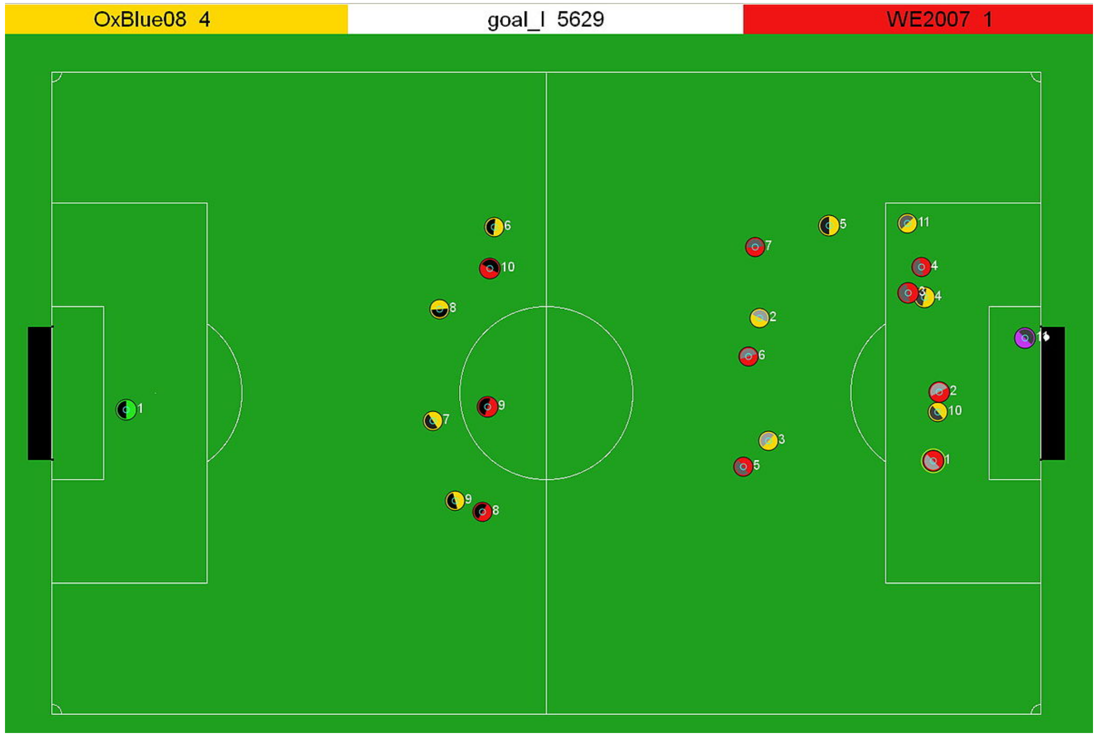
Planejamento de Caminho



Planejamento de Caminho



Planejamento de Ações no Futebol de Robôs



- Pense no agente que está com a bola.
- Objetivo: fazer gol.
- Ações possíveis: conduzir bola, driblar oponente, passe, chute a gol etc.
- Qual sequência de ações cooperativas para chegar no gol?

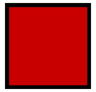
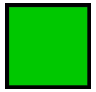
Modelagem do Problema de Busca

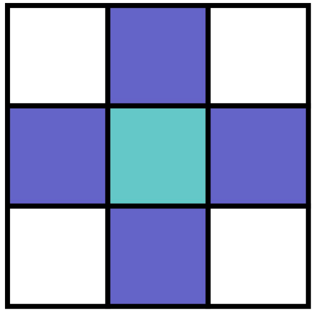
- Estados: situações possíveis do mundo.
- Ações: o que o agente pode fazer.
- Função sucessor: $s' = f(s, a)$.
- Estado inicial: estado onde se começa.
- Objetivo: estado onde se quer chegar.

Algoritmo de Busca

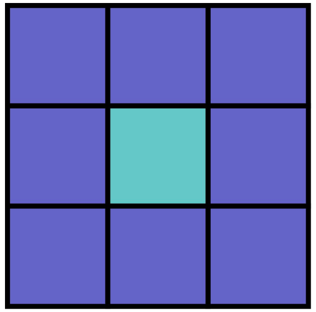
- Um algoritmo de busca explora os estados através da aplicação da função sucessor até atingir o objetivo.
- Constrói uma árvore (de busca).
- **Observação:** pode ser possível retornar a um estado já visitado durante a busca (mundo é um grafo).

Planejamento de Caminho

- Estados: posições no grid discretizado.
- Ações: movimentos (4-conectado ou 8-conectado).
- Função sucessor: posição após executar o movimento.
- Estado inicial: 
- Estado objetivo: 



4-conectado



8-conectado

Planejamento de Ações no Futebol de Robôs

- Estados: posições dos jogadores e da bola.
- Ações: conduzir a bola, driblar o oponente, passe, chute a gol etc.
- Função sucessor: posições dos jogadores e da bola após execução da ação (simulação).
- Estado inicial: situação atual de jogo.
- Estado objetivo: gol.
- Para uma modelagem mais fiel, é necessário um “modelo” do comportamento do oponente.

Planejamento x Controle

- Planejamento: determinar sequência de estados/ações até atingir o objetivo.
- Em geral, é muito custoso planejar usando ações de baixo nível (muito passos).
- Ser humano não planeja seu dia pensando no movimento de cada músculo.
- Uma abordagem comum em robótica é separar o planejamento do controle (execução).
- Também é comum haver vários níveis de planejamento (hierarquia).
- Planejamento: técnicas de IA (modelo simplificado de mundo).
- Controle: técnicas de teoria de controle (sistemas dinâmicos).

Planejamento x Controle

- Planejamento costuma ter dinâmica mais lenta, mas decisão requer algoritmo mais custoso (e.g. busca em grafo).
- Controle tem dinâmica rápida, mas decisão é simples (e.g. PID).
- Controle pressupõe malha fechada.
- Se o ambiente muda (dinâmico), então é necessário replanejar (planejamento em malha fechada).
- Planejamento e Controle: tema de CT-XXX (próxima disciplina).
- Na aula de hoje, vamos ver planejamento de computeiros (com busca em grafos).

Por que Controle?

- Robôs percebem o ambiente e executam ações imperfeitamente.
- Robôs não “andam reto”.
- Malha fechada envolve ler informações dos sensores e se corrigir a cada instante.
- Ideia: se desvio para a direita, devo mover a direção para a esquerda.

Planejamento de Caminho x Planejamento de Trajetória

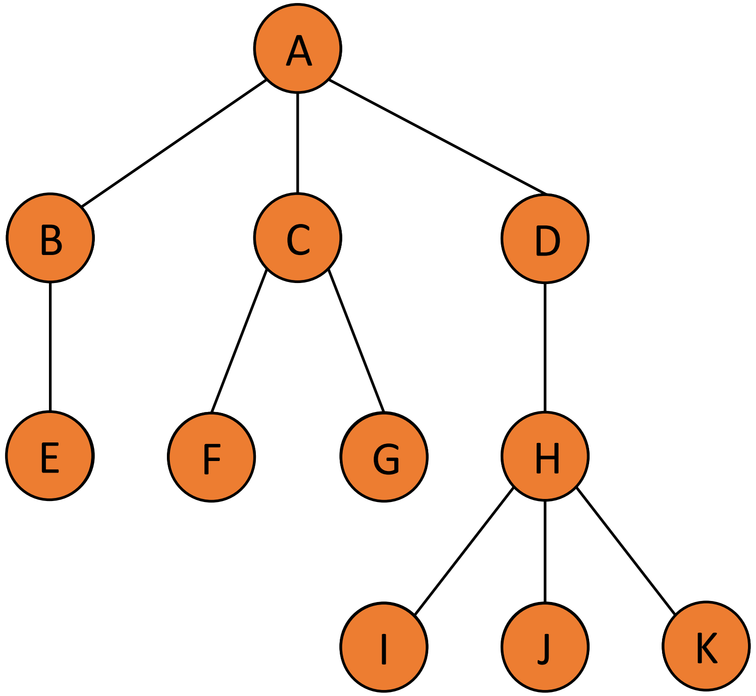
- Planejamento de caminho: (x, y, ψ).
- Planejamento de trajetória: (x, y, ψ, t).
- Técnicas da aula de hoje planejam caminho.
- Como executar no tempo é um passo posterior da IA.

Busca em Árvore

Busca em Árvore

- Depth-First Search (DFS): aprofunda primeiro e depois busca no mesmo nível;
  - Pré-ordem.
  - Pós-ordem.
  - In-ordem.
- Breath-First Search (BFS): busca mesmo nível primeiro e depois aprofunda;
  - Solução com caminho mínimo.
- Considerar:
  - n: número de elementos da árvore.
  - p: número de elementos no caminho até o objetivo.

Exploração em Profundidade (Pré-ordem)



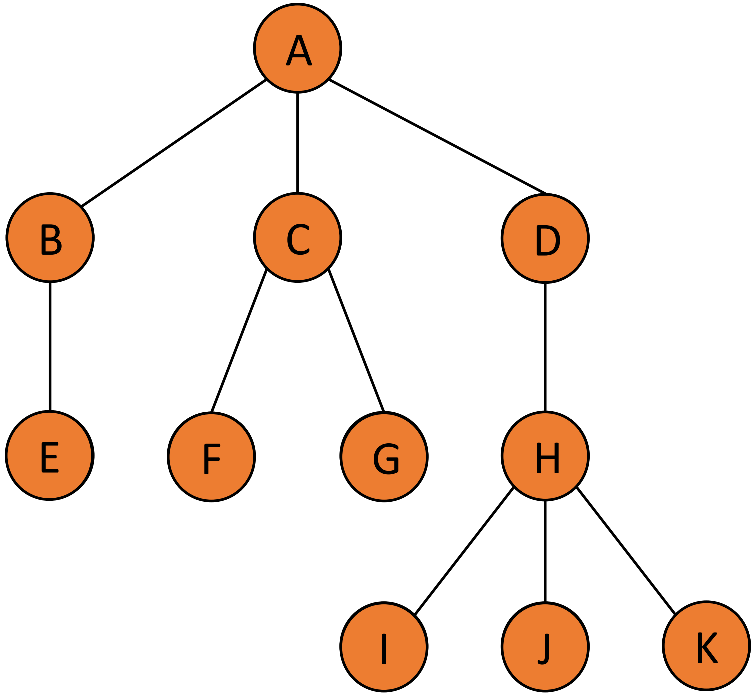
Exploração em Profundidade (Pré-ordem)

```
dfs(node):
   process(node)
   for child in node.children:
      dfs(child)
```

Observação: isso é “pseudocódigo”, **não** é Python ;).

Complexidade: $O(n)$

Exploração em Largura



Exploração em Largura

```
bfs(root):
  queue = Queue()
  queue.enqueue(root)
  while not queue.empty():
    node = queue.dequeue()
    process(node)
    for child in node.children:
      queue.enqueue(child)
```

Complexidade: $O(n)$

Busca em Largura

```
bfs(root, goal):
  queue = Queue()
  queue.enqueue(root)
  while not queue.empty():
    node = queue.dequeue()
    for child in node.children:
      if child.content == goal:
        return child
      queue.enqueue(child)
```

Complexidade: $O(n)$

Construir Caminho

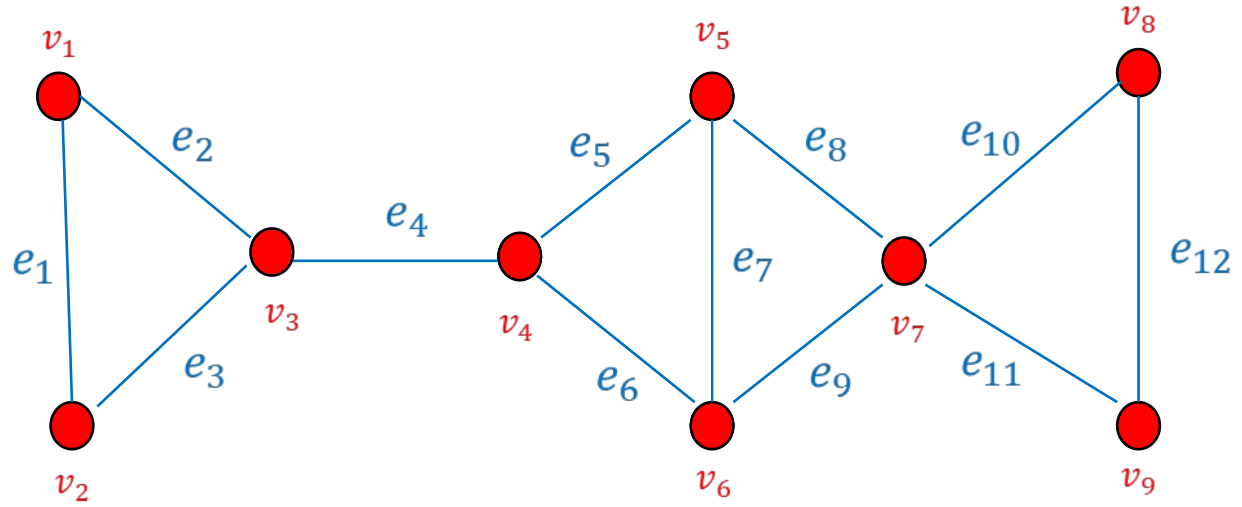
```
construct_path(goal):
   stack = Stack()
   node = goal
   while node is not None:
      stack.push(node)
      node = node.parent
   path = []
   while not stack.empty()
      path.append(stack.pop())
   return path
```

Complexidade: $O(p)$

Busca em Grafos

Revisão de Grafos

- $G = (V, E)$, em que $V = \{v_1, v_2, v_3, \dots, v_n\}$ e $E = \{e_1, e_2, e_3, \dots, e_m\}$.
- V é conjunto de nós e E é conjunto de arestas. $|V| = n$ e $|E| = m$.
- Exemplo:



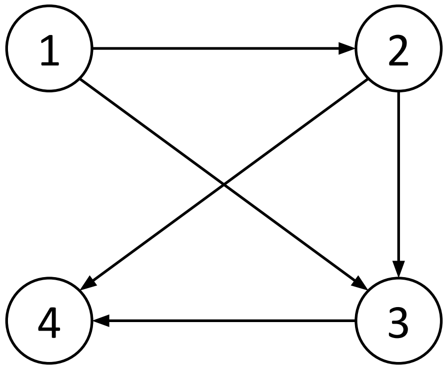
$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$ $\quad n = 9$

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\}$ $\quad m = 12$

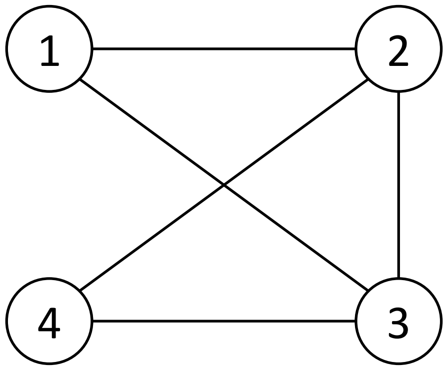
Grafos

- Orientado x Não orientado.
- Cíclico x Acíciclo.
- Directed Acyclic Graph (DAG): árvore.
- Pode conter informações nos nós ou nas arestas.

Matriz de Adjacências



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

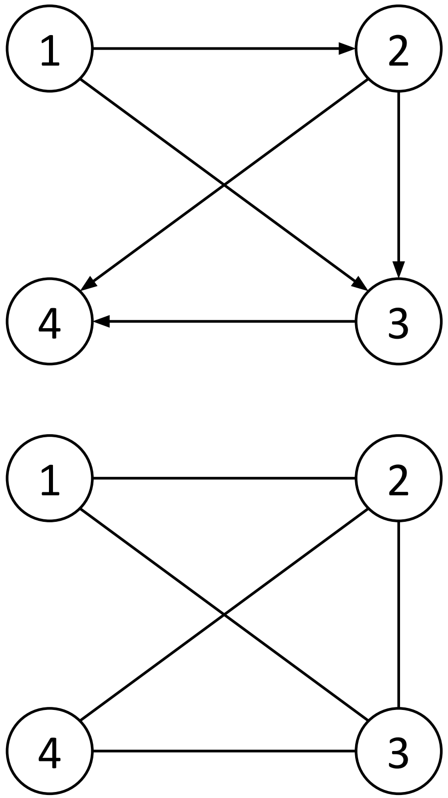


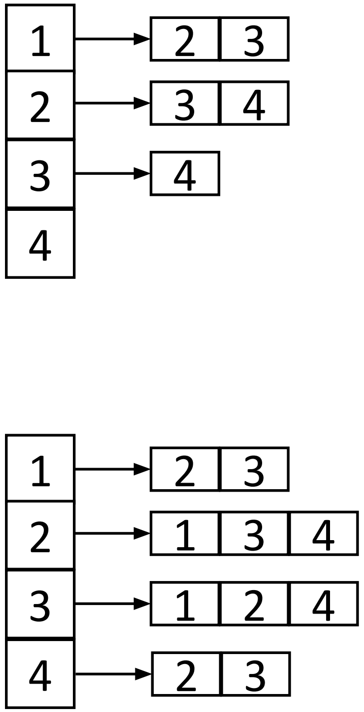
$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Memória: $O(n^2)$

Interessante quando grafo é **denso**

Lista de Adjacências





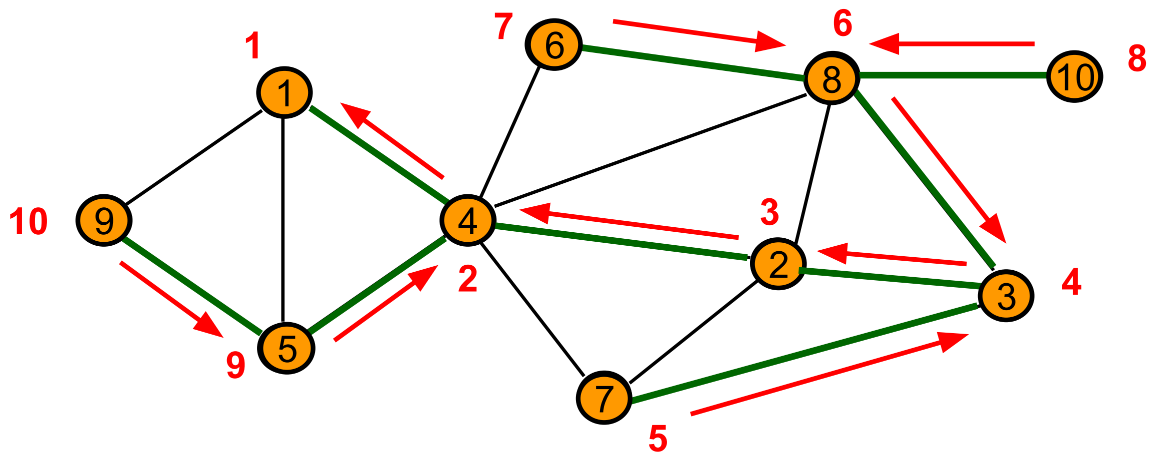
Memória: $O(n+m)$

Interessante quando grafo é **esparso**

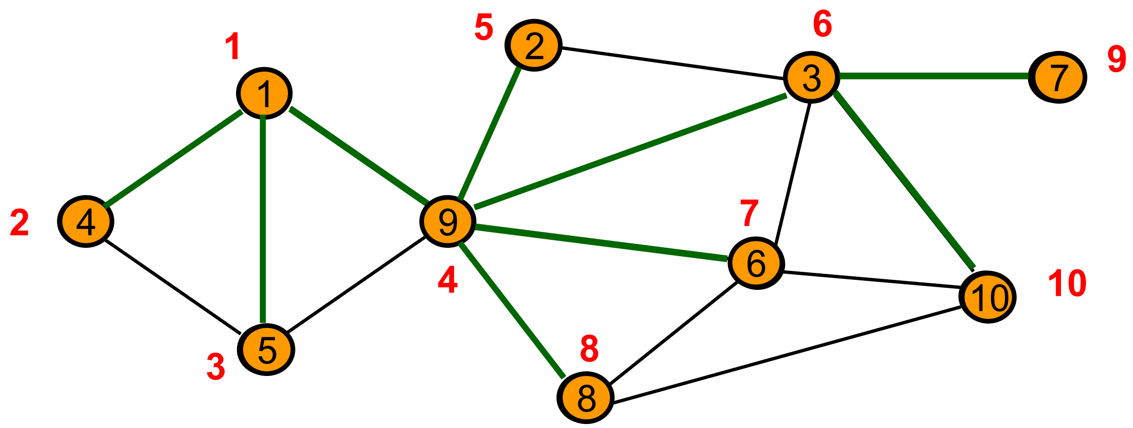
Busca em Grafos

- Depth-First Search (DFS): aprofunda primeiro e depois verificar no mesmo nível.
  - Várias aplicações: ordenação topológica, componentes fortemente conexos, vértices e arestas de corte.
- Breath-First Search (BFS): busca mesmo nível primeiro e depois aprofunda;
  - Solução clássica para problema de caminho mínimo.

Exploração em Profundidade



Exploração em Largura



Exploração em Largura

```
bfs(start):
    queue = Queue()
    queue.enqueue(start)
    exploration_number = 1
    start.visited = True
    start.exploration = exploration_number
    exploration_number += 1
    while not queue.empty():
        node = queue.dequeue()
        for successor in node.successors():
            if not successor.visited:
                successor.visited = True
                successor.exploration = exploration_number
                exploration_number += 1
                queue.enqueue(successor)
```

Complexidade: $O(n + m)$

Problema de Caminho Mínimo

- Problema muito importante de grafos.
- Aplicação clássica em robótica é para navegação de robôs móveis.
- Se custo de movimento é unitário, então BFS encontra solução ótima.
- BFS encontra caminho mínimo de uma origem até todos os demais vértices do grafo.
- Se custo não é unitário, mas é uniforme, BFS ainda é solução ótima.
- Em Robótica, em geral deseja-se caminho até certo objetivo, logo é comum parar a busca antes.

Busca em Largura

```
bfs(start, goal):
    queue = Queue()
    queue.enqueue(start)
    start.cost = 0
    start.visited = True
    while not queue.empty():
        node = queue.dequeue()
        for successor in node.successors():
            if not successor.visited:
                successor.cost = node.cost + 1 # unitary cost
                successor.visited = True
                successor.parent = node
                if successor.content == goal:
                    return successor, successor.cost
                queue.enqueue(successor)
```

Complexidade: $O(n + m)$

Algoritmo de Dijkstra

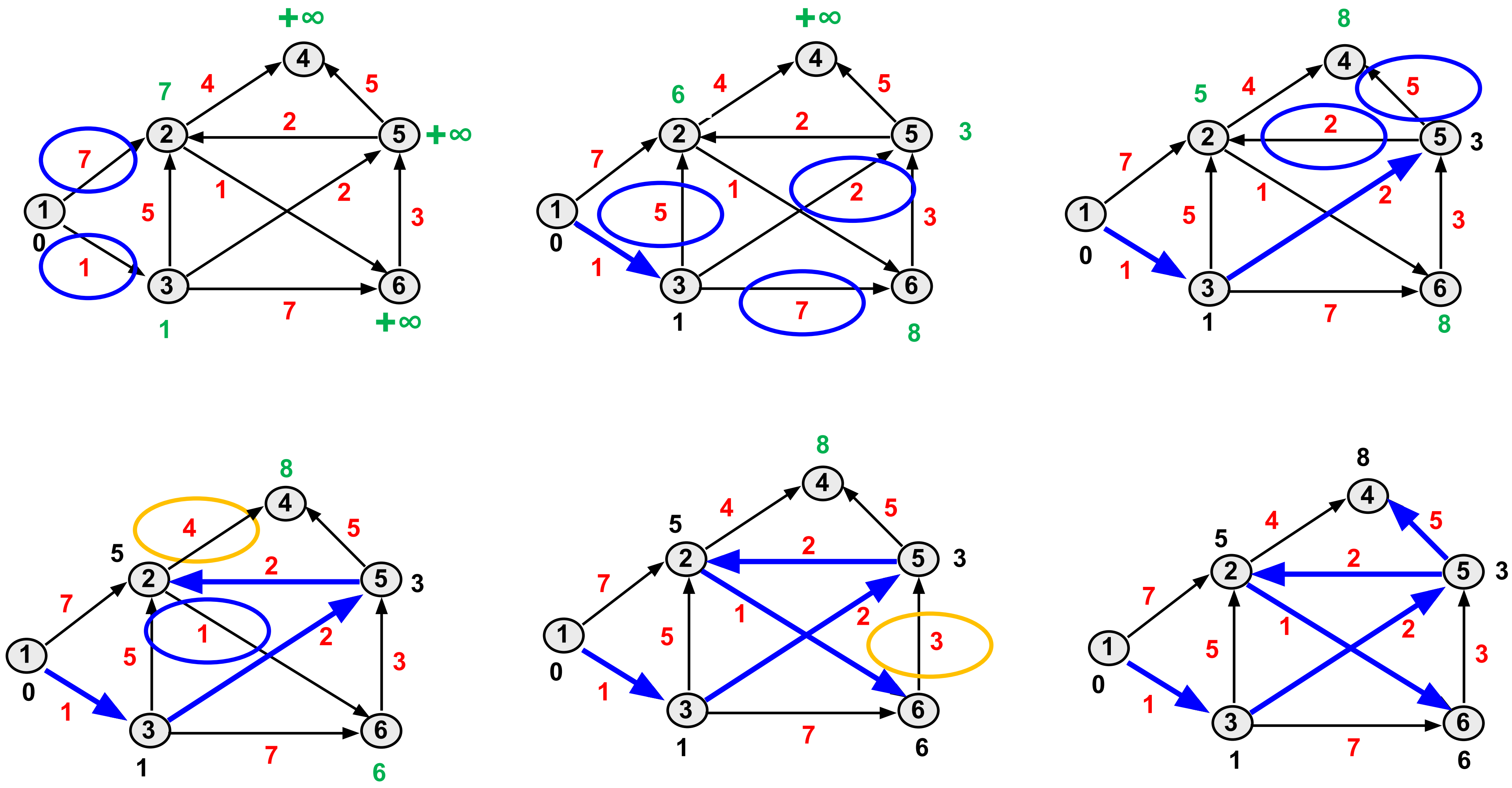
Algoritmo de Dijkstra

- Quando arestas possuem custo não uniforme, BFS não acha caminho ótimo.
- O algoritmo de Dijkstra generaliza a ideia de BFS para arestas com custos.
- Também encontra menores caminhos de origem até todos os demais vértices.

Algoritmo de Dijkstra

- Próximo elemento é o de menor distância desde a origem.
- Leva em conta que se pode encontrar caminho menor em vértice já visitado.
- Vértices podem estar em 3 estados:
  - Não visitado.
  - Em exploração (na fila).
  - Explorado (fora da fila).
- Se o nó foi "explorado", já se sabe sua distância minima.
- Para eficiência, é importante usar uma *fila de prioridades*.

Exemplo de Dijkstra



Fila de Prioridades (de Mínimo)

- Estrutura de dados com seguintes operações:
  - `extract_min()`: retorna o elemento mínimo e o remove.
  - `insert_or_update(value, element)`: insere um novo elemento ou atualiza (se já estiver na fila de prioridades).
- Também existe a estrutura equivalente de máximo.
- Implementação padrão usa *heap*.
- Se corretamente implementada, complexidade das operações é $O(logn)$.

Algoritmo de Dijkstra

```
dijkstra(start):
    # Initialize node.cost to inf for all nodes
    pq = PriorityQueue()
    start.cost = 0
    pq.insert_or_update(start.cost, start)
    while not pq.empty():
        node = pq.extract_min()
        for successor in node.successors():
            if successor.cost > node.cost + cost(node, successor):
                successor.cost = node.cost + cost(node, successor)
                successor.parent = node
                pq.insert_or_update(successor.cost, node)
```

Complexidade:
Com fila de prioridades: $O((n+m)logn)$
Sem fila de prioridades: $O(n^2)$

Observação: quando o nó é retirado da fila, já se tem certeza que o custo mínimo foi determinado.

Para busca, parar quando objetivo for retirado da fila

Busca Informada

Busca Informada

- Dijkstra acha solução ótima, mas visita muitos nós.
- É possível acelerar a busca usando “conhecimento do domínio” (informação).
- Considerar primeiro nós mais promissores: *best-first search*.
- Para isso, usa-se estimativa de quão promissor é certo nó.
- Usa-se uma *função de avaliação heurística* para estimar a “promessa” de um nó.
- Para caminho mínimo, usa-se uma estimativa de custo até o objetivo.
- Ideia aplicável para busca em árvore ou grafo.

Busca Informada

Observação: aqui não estamos nos preocupando com repetição de nós

```
best_first_search(start, goal):
    pq = PriorityQueue()
    pq.insert_or_update(start.evaluate(), start)
    while not pq.empty():
        node = pq.extract_best() # best refers to min or max
        for successor in node.successors():
            successor.parent = node
            if successor.content == goal:
                return successor
            pq.insert_or_update(successor.evaluate(), successor)
```

Busca Informada

- Sejam:
  - $g(n)$: custo para chegar até o nó n.
  - $h^*(n)$: custo ótimo do nó n até o objetivo.
  - $f^*(n) = g(n) + h^*(n)$.
  - $h(n)$: função heurística para estimar $h^*(n)$.
  - $f(n) = g(n) + h(n)$.
- Para o caso de minimizar caminho, tem-se 3 tipos de busca:
  - Busca de custo uniforme: usa $g(n)$, i.e. custo do caminho até o nó em questão.
  - Busca gulosa (*greedy*): usa $h(n)$, i.e. apenas estimativa do custo até o objetivo.
  - A*: usa $f(n)$, i.e. estimativa de custo total do nó inicial até o objetivo.
- Busca gulosa é rápida, mas pode encontrar solução subótima.
- A* melhora desempenho e é ótimo se $h(n)$ atender a certas condições.

Busca Gulosa

```
greedy_search(start, goal):
    pq = PriorityQueue()
    start.cost = h(start, goal)
    pq.insert_or_update(start.cost, start)
    while not pq.empty():
        node = pq.extract_min()
        for successor in node.successors():
            successor.parent = node
            if successor.content == goal:
                return successor
            successor.cost = h(successor, goal)
            pq.insert_or_update(successor.cost, successor)
```

A*

```
a_star(start, goal):
    # Initialize node.g and node.f to inf for all nodes
    pq = PriorityQueue()
    start.g = 0
    start.f = h(start, goal)
    pq.insert_or_update(start.f, start)
    while not pq.empty():
        node = pq.extract_min()
        if node.content == goal:
            return successor
        for successor in node.successors():
            if successor.f > node.g + cost(node, successor) + h(successor, goal):
                successor.g = node.g + cost(node, successor)
                successor.f = successor.g + h(successor, goal)
                successor.parent = node
                pq.insert_or_update(successor.f, successor)
```

Heurística

- Heurística depende do problema (conhecimento de domínio).
- Heurística **admissível**: $h(n) \leq h^*(n)$. Intuivamente, não superestima o custo real (estimativa otimista).
- Heurística **admissível** garante **A* ótimo** para busca em **árvore**
- Heurística **consistente**: $h(n) \leq c(n, a, n') + h(n')$, em que n' é sucessor de n através de a. Intuitivamente, isso diz que a heurística respeita a "desigualdade triangular".
- Heurística **consistente** garante **A* ótimo** para busca em **grafo**.
- Para desempenho, consistente é melhor que admissível.
- Consistente implica admissível, mas não o contrário.

Heurística

- Na prática, é difícil conseguir heurística admissível e não consistente. Maioria dos exemplos são “forçados”.
- Quanto mais $h(n)$ for próximo de $h^*(n)$, melhor.
- Se $h_1(n) \geq h_2(n), \forall n$, diz-se que h_1 domina h_2. Pode-se mostrar que A* sempre expande menos nós usando h_1 do que usando h_2.

Como Encontrar uma Heurística?

- "Criatividade".
- Dica: pensar no problema "relaxado".
- Para problemas de planejamento de caminho, usar distância euclidiana:

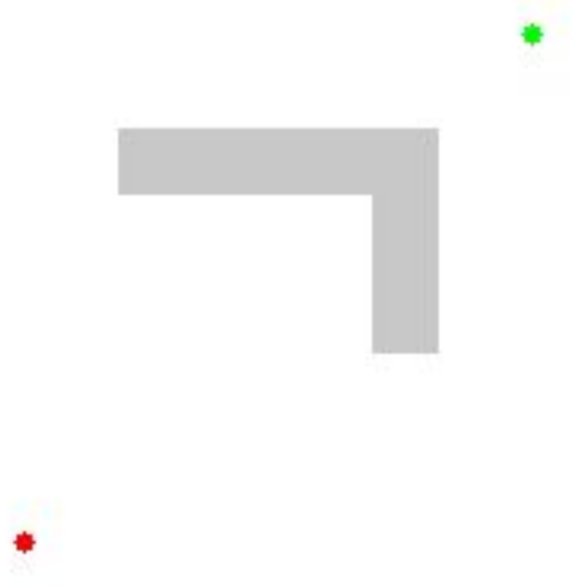
$$h(n, g) = \sqrt{(n.x - g.x)^2 + (n.y - g.y)^2}$$

- Se 4-conectado, usar distância de Manhattan:

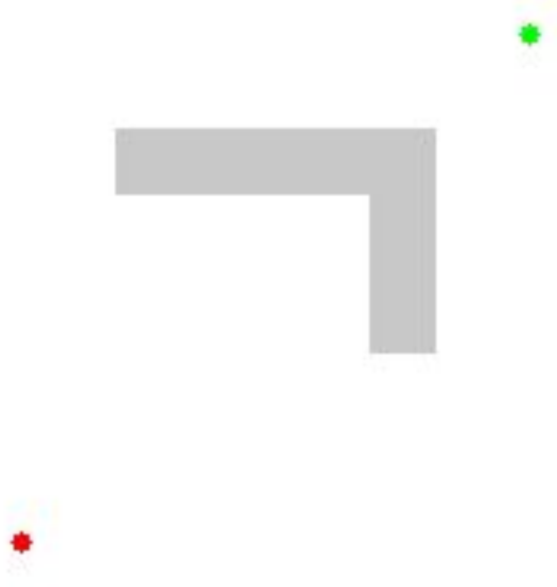
$$h'(n, g) = |n.x - g.x| + |n.y - g.y|$$

Dijkstra x A*

Dijkstra



A*



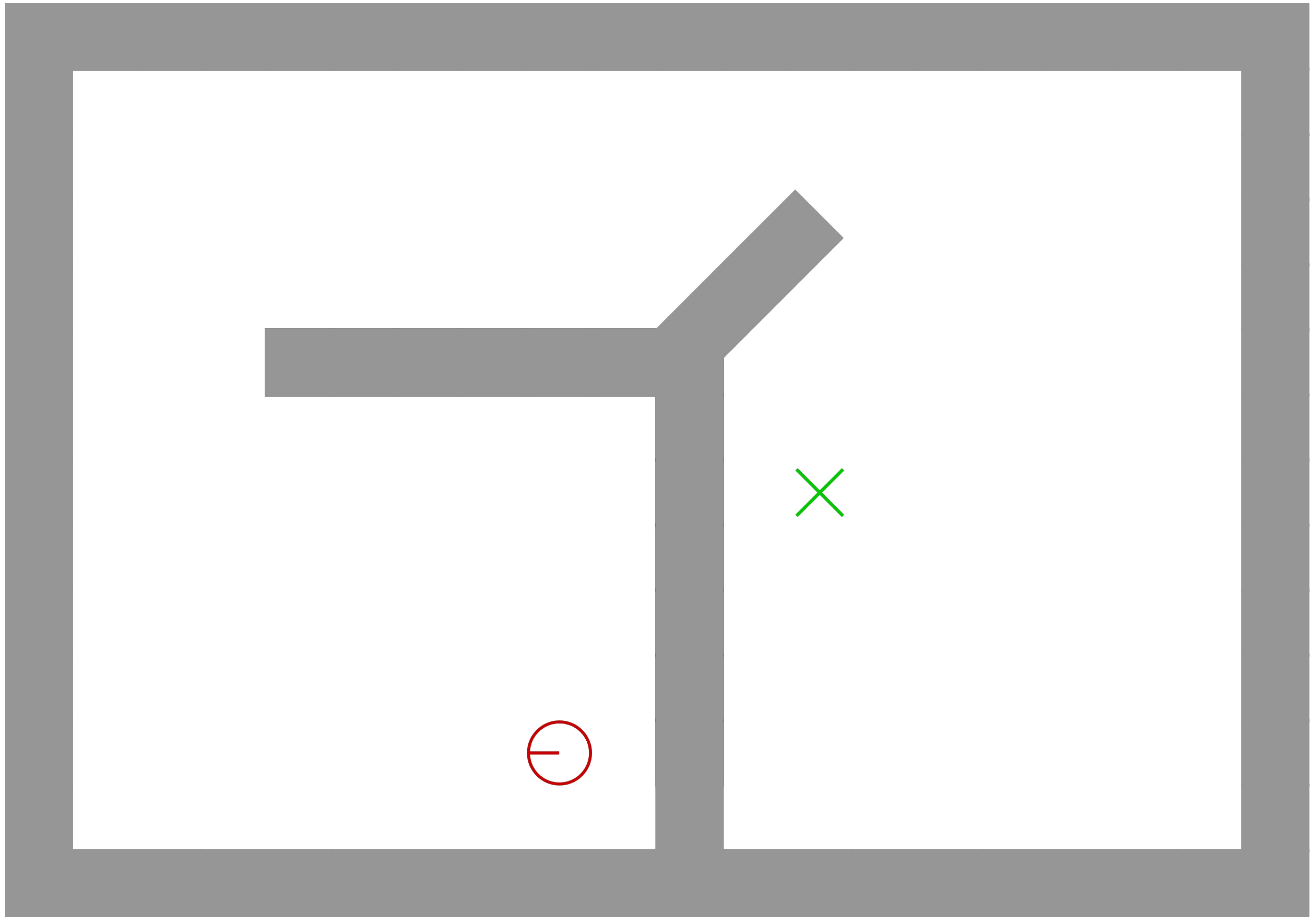
Fonte: https://en.wikipedia.org/wiki/A*_search_algorithm

Geração de Grafos para Planejamento de Caminho

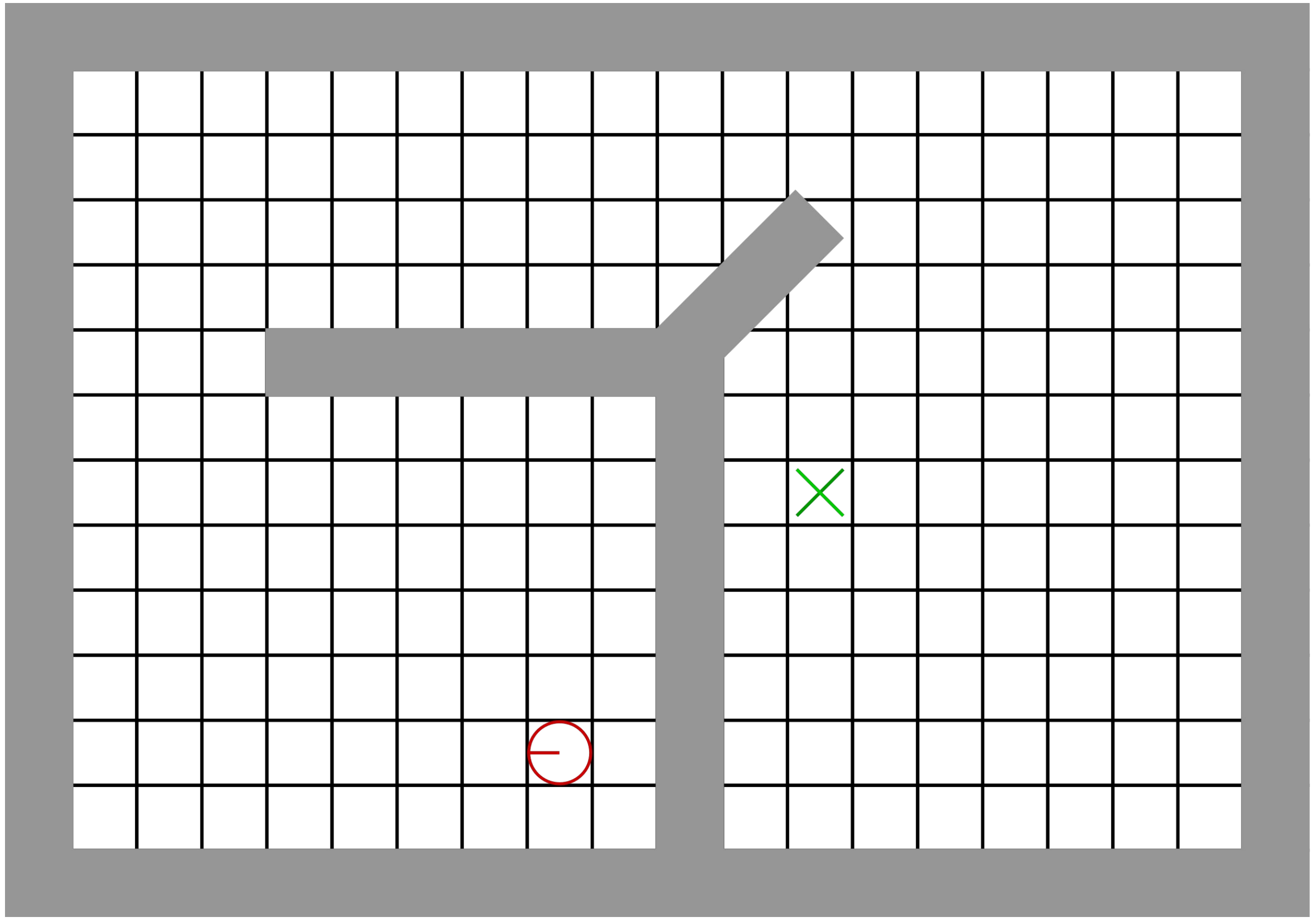
Geração de Grafos

- Para aplicar A*, é necessário gerar um grafo que representa o mapa.
- Considera-se robô pontual: aumentar tamanho dos obstáculos.
- Costuma-se adicionar uma margem de segurança nos obstáculos para acomodar erros de execução do caminho.
- Necessário reconstruir o grafo se o ambiente for dinâmico.

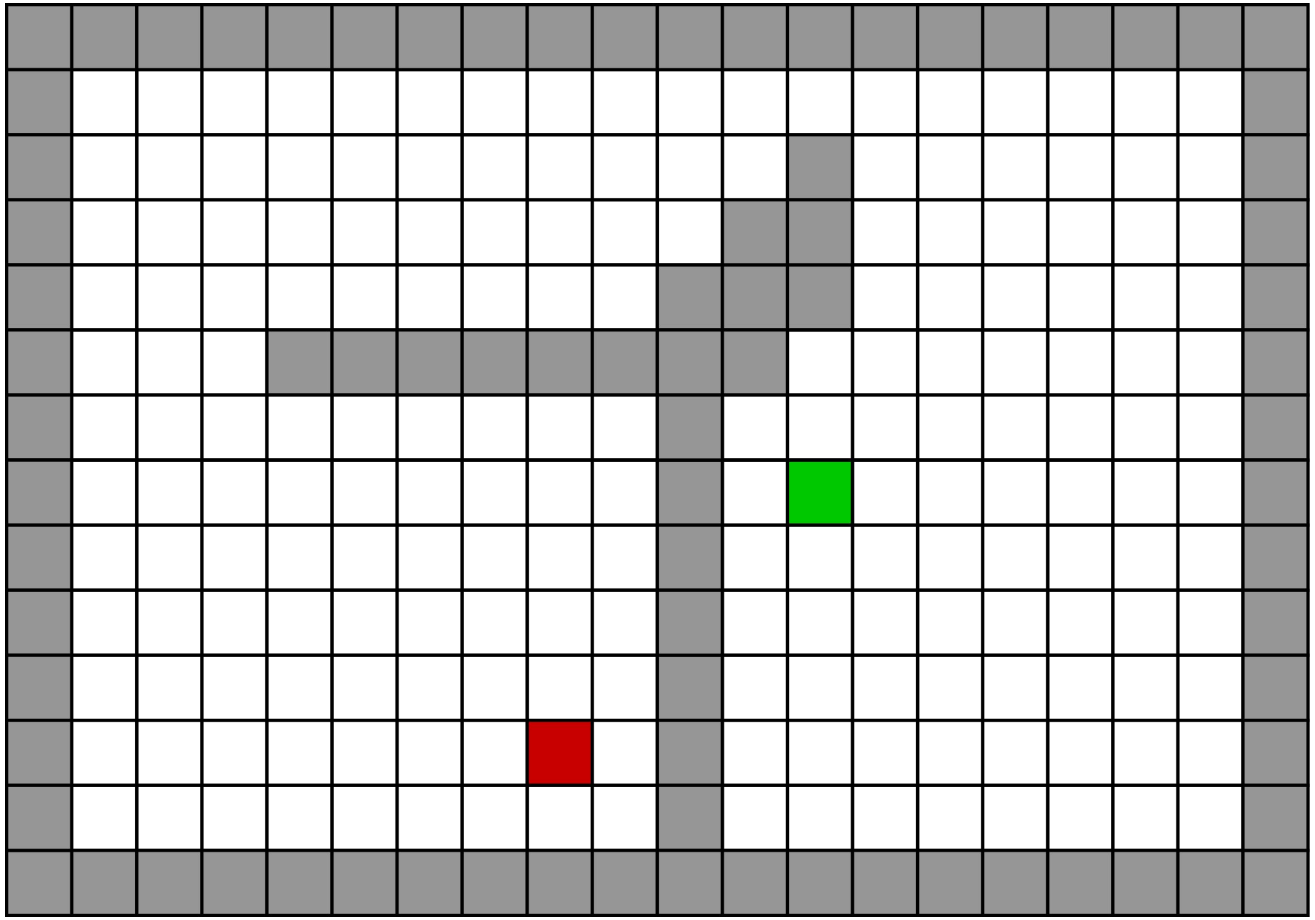
Grid de Ocupação



Grid de Ocupação



Grid de Ocupação



Grid de Ocupação

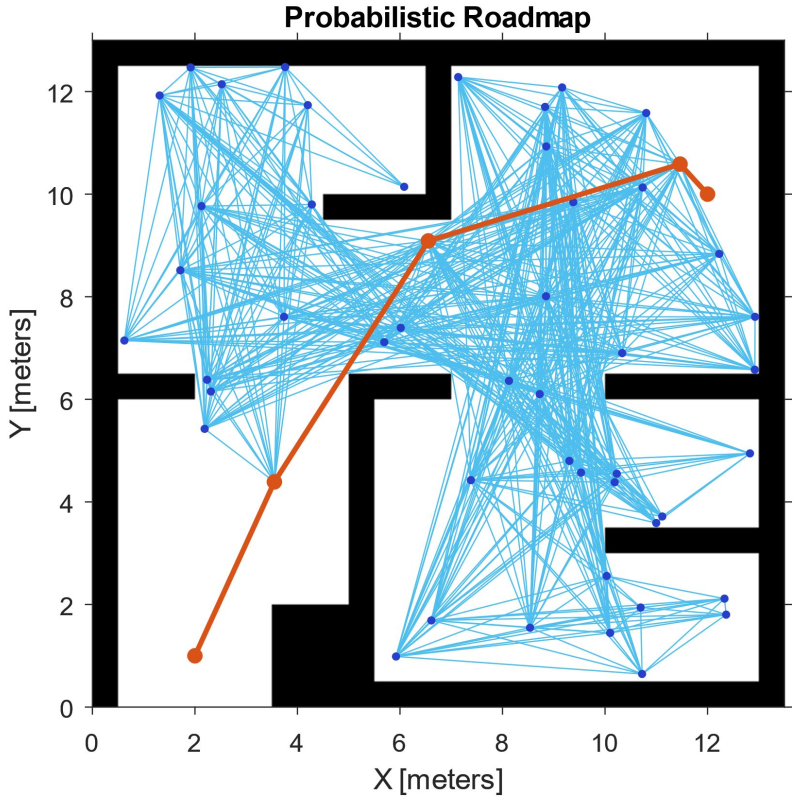
- Por otimização, não se constroi o grafo explicitamente, já que fica subentendido (4-conectado ou 8-conectado).
- Resolução do *grid* é um *trade-off* entre precisão e custo computacional.
- Quando se usa 8-conectado, costuma-se usar fator de $\sqrt{2}$ para custo de movimento em diagonal.
- Permite considerar custo do terreno, e.g. custo para andar na água é maior do que para andar em terreno plano (muito usado em jogos).
- Em robótica, é comum considerar custo maior em células próximas a obstáculos (tentar gerar caminho seguro).

Grid de Ocupação

- Devido à discretização, a solução é “subótima”.
- É possível usar resolução variável do *grid*.
- Generalização: *cell decomposition*.
- **Não** é completo: pode falhar devido à resolução do *grid* (na prática dificilmente é problema).
- É possível considerar rotação, mas aumenta dimensão do espaço de estados.
- Para robótica, a pior desvantagem é a geração de rotações bruscas (quinas).

Probabilistic Roadmap

1. Amostrar aleatoriamente o espaço.
2. Descartar amostras que caem dentro de obstáculos.
3. Adicionar nós inicial e objetivo.
4. Ligar vértices que possuem visibilidade.
5. Determinar caminho mínimo com A*.

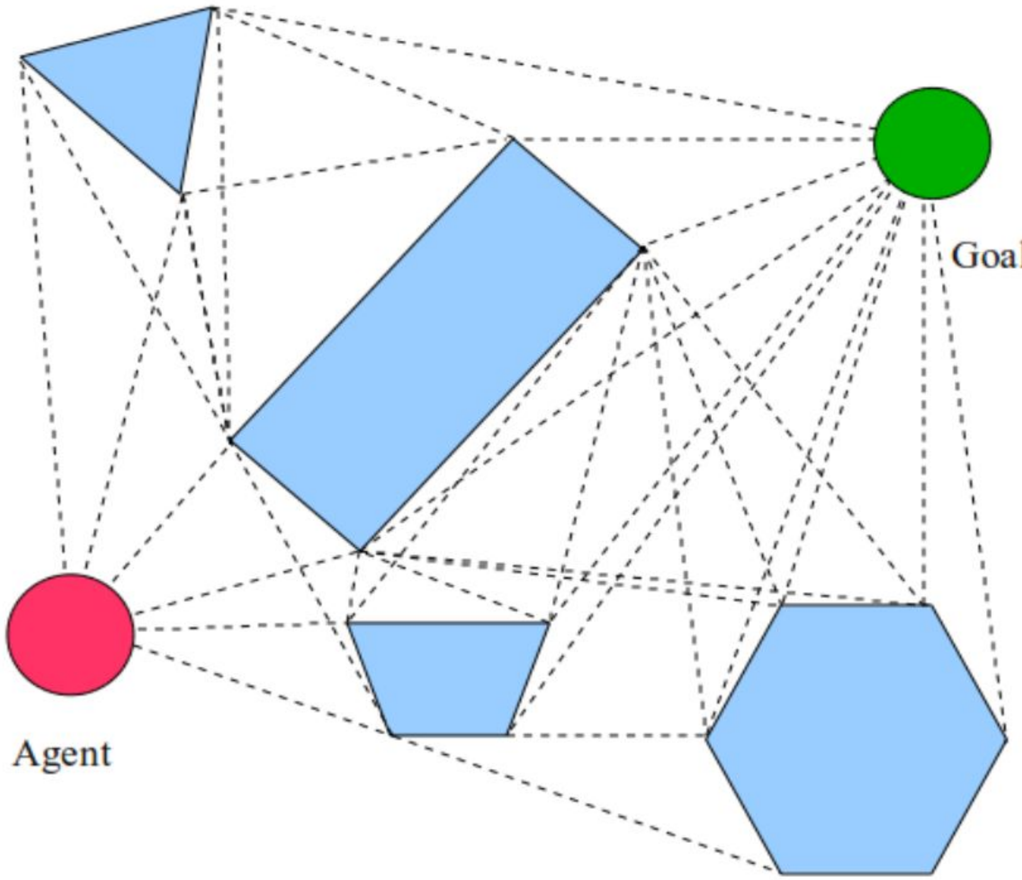


Probabilistic Roadmap

- Método baseado em amostragem.
- Subótimo devido à amostragem.
- Caminho diferente a cada execução (não-determinístico).
- Número de amostras determina *trade-off* entre precisão e custo computacional.
- *Roadmap* é caro de construir, logo costuma ser feito *off-line*.
- Costuma-se limitar número de arestas (e.g. ligar com k vizinhos mais próximos).
- Funciona bem na prática.
- **Não** é completo: pode falhar devido à amostragem.

Visibility Graph

1. Incluir vértices dos obstáculos como vértices do grafo.
2. Incluir nós inicial e objetivo.
3. Ligar vértices que possuem visibilidade.
4. Determinar caminho mínimo com A*.



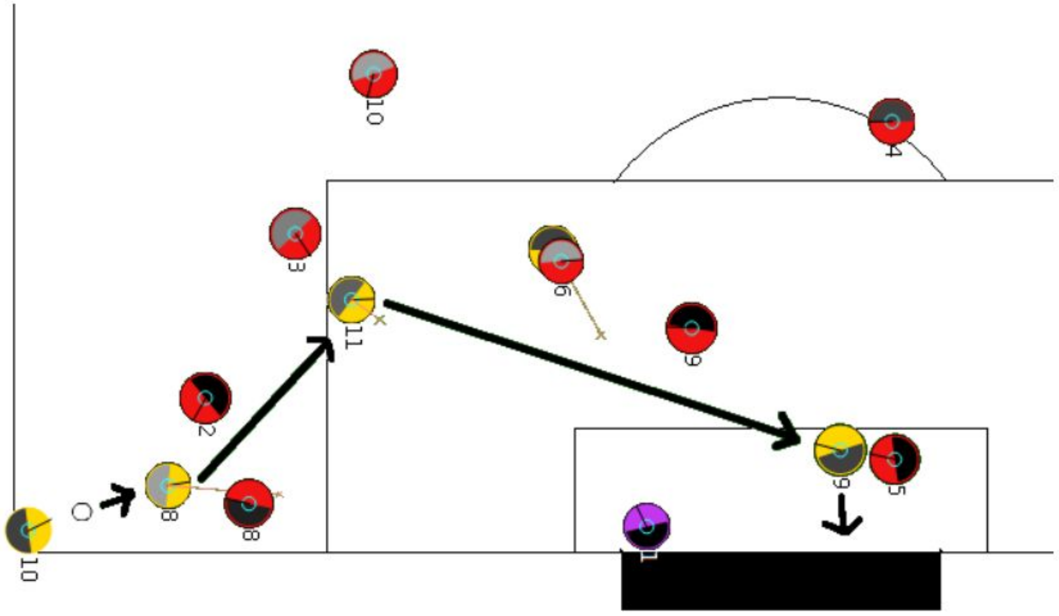
Visibility Graph

- Se robô conseguir se mover em linha reta, encontra solução ótima.
- Custo computacional para construir grafo de visibilidade é alto, logo costuma ser feito *off-line*. Truque: construir à medida que planeja.
- Desvantagem: “cola” nos obstáculos (pouco seguro).

Planejamento de Ações no Futebol de Robôs

Action Chain Framework

- Planejamento cooperativo de ações para robô que está com a bola.
- Desenvolvido pelo time japonês HELIOS (Hidehisa Akiyama).



Action Chain Framework

- Cooperativo: decido pelo meu companheiro (funciona pois ele executa mesmo código).
- Busca em árvore (repetição de estados improvável).
- Execução usa comunicação entre os agentes.
- Estado: situação da campo (posições dos jogadores e da bola).
- Ações: conduzir a bola, passes (vários tipos), chute a gol.
- Busca *greedy*: explora primeiro nó mais promissor.
- Ignora custo do caminho.

Action Chain Framework

- Heurística usa combinação linear de *features*:
  - Coordenada x da bola.
  - Distância da bola até gol adversário.
  - Distância da bola até nosso gol.
  - Distância da bola até oponentes.
  - Situações especiais: bola fora, bola no nosso gol, bola no gol adversário etc.
- Versão original considera oponentes parados.
- Mundo incerto: exploração com profundidade limitada.
- Limitação de número de nós explorados.
- Funciona muito bem na prática!

Para Saber Mais

- Busca informada usando grafos: capítulos 3 e 4 do livro Inteligência Artificial de Russell & Norvig.
- Planejamento de caminho/trajetória: capítulo 6 do livro Autonomous Mobile Robots (2nd edition) de Siegwart, Nourbakhsh e Scaramuzza.
- Bíblia de planejamento: Steven M. LaValle. Planning Algorithms. Cambridge University Press. 2006.
  - Link: http://planning.cs.uiuc.edu/
- Action chain: Akiyama, H.; Nakashima, T. Online Cooperative Behavior Planning using a Tree Search Method in the RoboCup Soccer Simulation.

Vídeos Legais :)

- A* no Darpa Urban Challenge: https://www.youtube.com/watch?v=qXZt-B7iUyw
- Navegação em jogos: https://www.youtube.com/watch?v=U5MTIh_KyBc

Laboratório 2

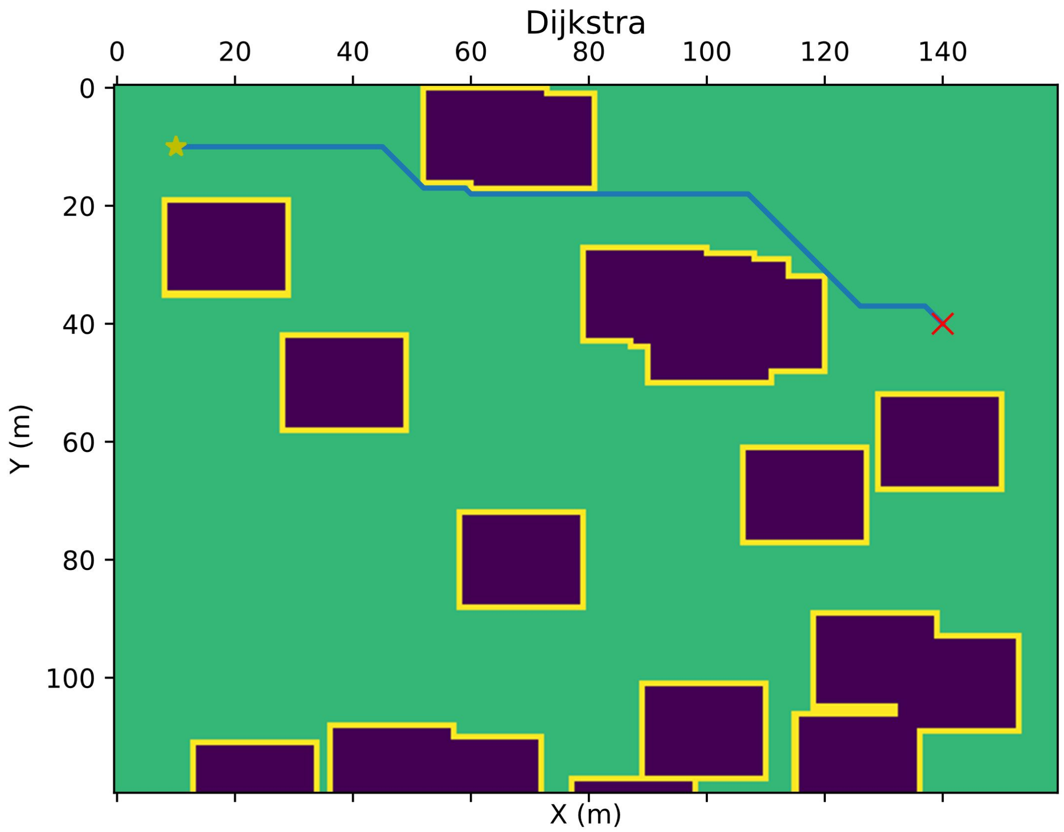
Laboratório 2

- Implementar planejamento de caminho em grid usando:
  - Dijkstra.
  - Greedy Best-First.
  - A*.
- Comparar as implementações em termos de tempo computacional e custo do caminho.
- 8-conectado.
- Custo de movimento em diagonal é $\sqrt{2}$.
- Custo maior na proximidade de obstáculos.

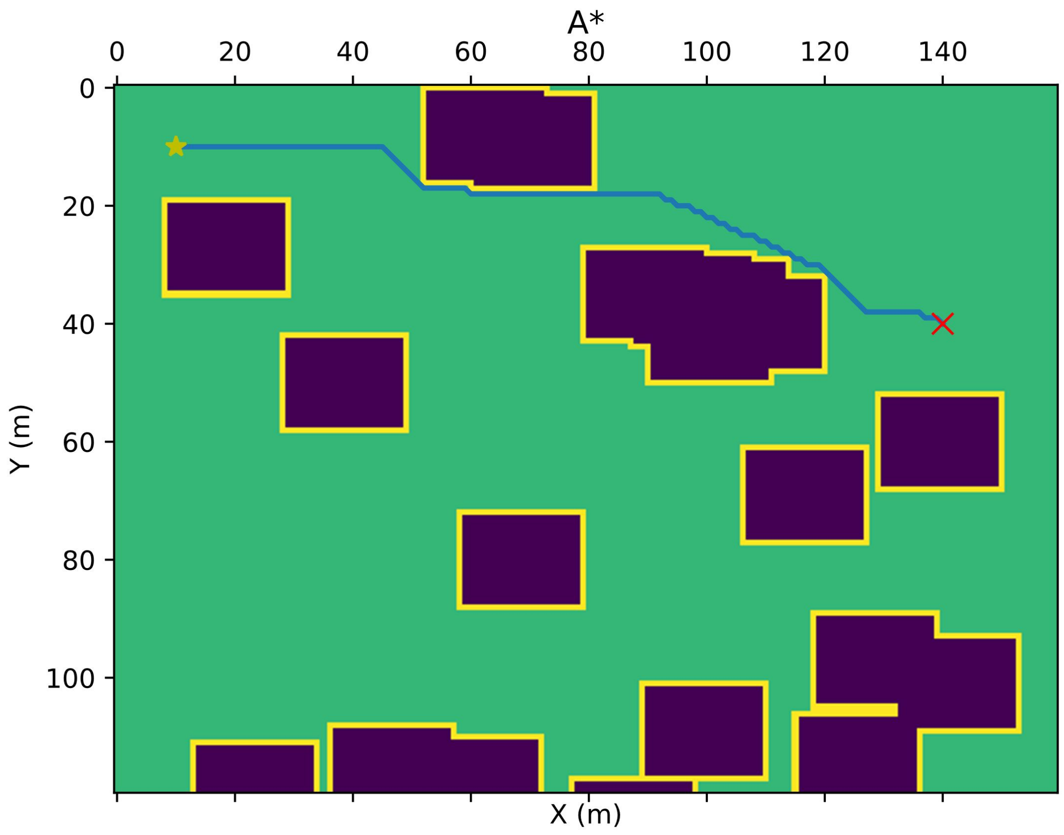
Laboratório 2

- Uso de *heap* do Python:
  - Criação: `pq = []`
  - Inserção: `heapq.heappush(pq, (node.f, node))`
  - Extração: `f, node = heapq.heappop(pq)`
- **Atenção:** não tem como atualizar! Assim, pode inserir mais de uma vez.
- Solução: se já foi retirado alguma vez, ignora. Usar `node.closed`.
- É feio, mas não aumenta complexidade.

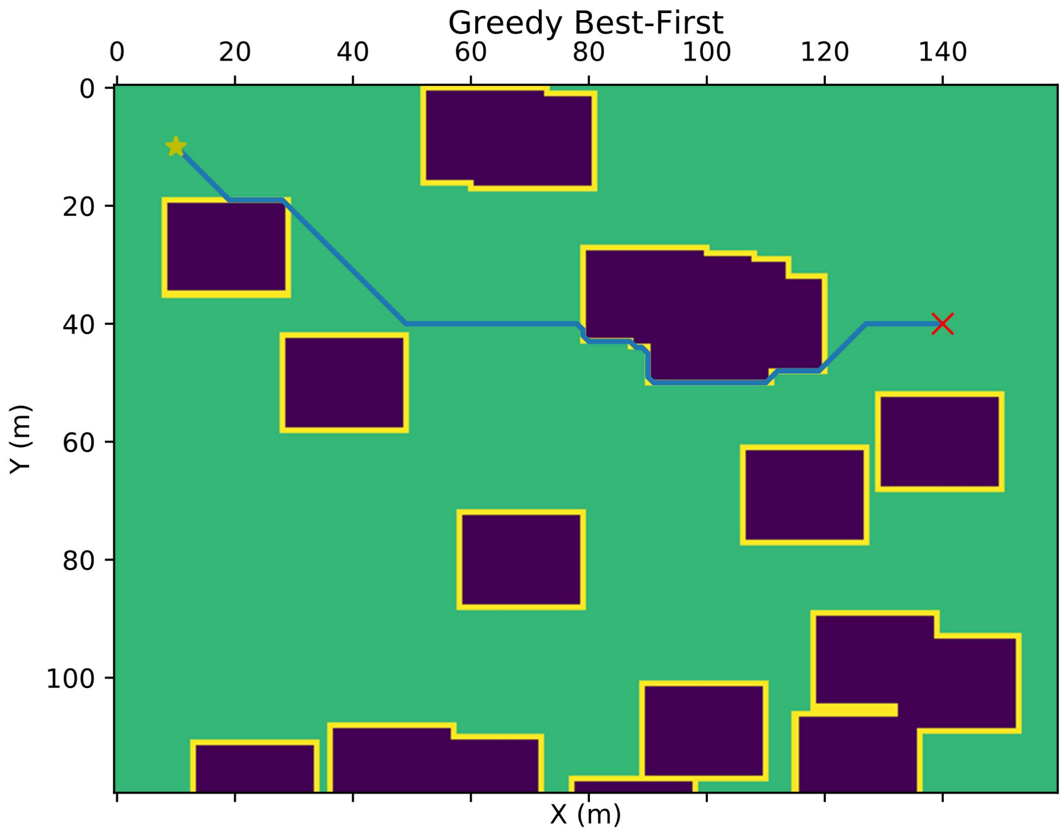
Caminho com Dijkstra



Caminho com A*



Caminho com Greedy Best-First



Comparação

Algoritmo	Custo do Caminho	Tempo Computacional (s) *
Dijkstra	142,4264	0,3221
A*	142,4264	0,0864
Greedy	215,7817	0,0105

* Processador: Intel Core i7-6700HQ @ 2.6 GHz