Capítulo 1

- A inteligência está preocupada principalmente com a ação racional. Idealmente um agente toma a melhor ação possível numa situação. Estudar o problema de construir agentes inteligentes nesse sentido.
- A inteligência artificial avançou mais rapidamente na última década devido ao maior uso do método científico na experimentação e comparação de abordagens.
- O progresso recente na compreensão da base teórica da inteligência foi acompanhado de melhorias nas capacidades dos sistemas reais. Os subcampos da inteligência artificial tornaram-se mais integrados e a IA encontrou um terreno comum com outras disciplinas.

Capítulo 2

- Um agente é algo que percebe e age num ambiente. A função agente especifica a ação executada pelo agente em resposta a qualquer sequência de perceções.
- A medida de desempenho avalia o comportamento do agente num ambiente. Um agente racional atua de modo a maximizar o valor esperado da medida de desempenho, dada a sequência percetiva que viu até ao momento.
- Uma especificação do ambiente de tarefas inclui a medida de desempenho, o ambiente externo, os atuadores e os sensores. Ao projetar um agente, a primeira etapa deve sempre ser especificar o ambiente da tarefa da forma mais completa possível.
- Os ambientes das tarefas variam ao longo de várias dimensões significativas. Podem ser totais ou parcialmente observáveis, agente único ou multiagente, determinístico ou estocástico, episódico ou sequencial, estático ou dinâmico, discreto ou contínuo e conhecido ou desconhecido.
- O programa do agente implementa a função do mesmo. Existe uma variedade de designs básicos de programas de agentes que refletem o tipo de informação tornada explícita e usada no processo de decisão. Os designs variam em eficiência, compactação e flexibilidade. O design apropriado do programa do agente depende da natureza do ambiente.
- Os reflexos simples dos agentes respondem diretamente às perceções, enquanto os agentes reflexivos baseados em modelos mantêm o estado interno para rastrear aspetos do mundo que não são evidentes na perceção atual. Agentes baseados em metas agem para atingir os seus objetivos, e agentes baseados em serviços públicos tentam maximizar a própria “felicidade” esperada.
- Todos os agentes podem melhorar o seu desempenho através da aprendizagem.

Capítulo 3

BFS

Pseudo-código:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

- É completo, é ótimo se todas as ações tiverem o mesmo custo, guarda todos os nós num set de explorados, todas as camadas de nós na profundidade máxima têm de ser expandidas antes de objetivo ser alcançado.
- Complexidade espacial: $O(b^d)$, d é a distância da solução mais próxima.
- Complexidade temporal: $O(b^d)$.

Uniform-Cost Search

Pseudo-código:

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

- É ótimo mesmo que o custo das ações seja diferente (devido à PriorityQueue), o objetivo alcançado é testado quando um nó é escolhido para expansão. O primeiro nó gerado poderá ser um caminho sub-ótimo. Um teste é adicionado no caso em que um caminho é encontrado para um nó que já se encontra na fronteira.
- Complexidade espacial: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, em que C* é o custo da solução ótima e assumir que cada ação custa pelo menos ϵ.
- Complexidade temporal: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

DFS & DLS

Pseudo-código:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

- Expande sempre o nó mais profundo. À medida que os eles são expandidos, são removidos da fronteira, para que a procura recue para o próximo nó mais profundo. Usa uma LIFO queue. É completo num espaço finito de estados pois vai eventualmente expandir todos os nós. Não é ótimo. Se usarmos backtracking, o algoritmo usa menos memória).
- Complexidade espacial: O(bm) m é a profundidade máxima de qualquer nó.
- Complexidade temporal: $O(b^m)$.

IDDFS

Pseudo-código:

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

- É completo quando o fator de expansão é finito e ótimo quando o custo de movimento é uma função crescente da profundidade do nó. Combina os benefícios de DFS e BFS.
- Complexidade espacial: O(bd).
- Complexidade temporal: $O(b^d)$.

Bidiretional Search

- A ideia da busca bidirecional é percorrer duas buscas em simultâneo. Uma desde o estado inicial e a outra a partir do estado final, esperando que as duas buscas se encontram no meio.
- Complexidade espacial: $O(b^{d/2})$.
- Complexidade temporal: $O(b^{d/2})$.

Greedy BFS

- Tenta sempre expandir o nó que está mais próximo do objetivo, no prossuposto de que nos irá levar a uma solução mais rapidamente. Avalia os seus nós a usar apenas a função heurística que é: *f(n) = h(n).*
- É incompleto até num espaço finito de estados.
- Complexidade espacial: $O(b^m)$, m é a máxima profundidade do espaço de procura.
- Complexidade temporal: $O(b^m)$.

A*

- Avalia os nós a combinar g(n), custo para alcançar um nó, e *h(n),* custo do nó para chegar ao objetivo*: f(n) = g(n) + h(n).*
- *f(n)* é o custo estimado da solução menos custosa por *n*.
- A* é completo e ótimo, o algoritmo é idêntico ao Uniform-Cost search, exceto que A* usa *g+h* em vez de *g.*
- A* é otimamente eficiente para qualquer heurística dada que seja consistente.
- A* não é prático para problemas de grande escala.

Heurística

- A condição que é requerida para a otimalidade é que *h(n)* seja uma heurística admissível. Uma heurística é admissível quando não sobrestima o custo para atingir o objetivo, elas são por natureza otimistas porque pensam que o custo de resolver o problema é mais pequeno do que realmente o é. Outra condição é ser consistente, ou seja, para qualquer nó *n* e para qualquer sucessor *n'* de n gerado por uma ação *a*, o custo estimado para atingir o objetivo desde *n* nunca é maior do que o custo de movimento de chegar a *n'* mais o custo estimado do objetivo até *n'*.

Memory-bounded heuristic search

- A forma mais simples de reduzir os requisitos de memória do A* é adaptá-lo á ideia de aprofundamento iterativo para o contexto de procura heurístico, resultando no iterative-deepening A* (IDA*). O *cutoff* do algoritmo usa o *f-cost(g+h)* em vez da profundidade. Em cada iteração, o valor de *cutoff* é o menor *f-cost* de qualquer nó que excede o *cutoff* da iteração anterior. IDA* é prático para problemas com custo unitário de movimento.
- Recursive best-first search (RBFS) é um algoritmo recursivo simples que tenta imitar a operação do bfs normal, mas a usar tempo linear. A estrutura é parecida com o DFS recursivo, mas em vez de continuar indefinidamente a descer pelo caminho corrente, usa a variável *f_limit* para manter rastreio do *f-value* do melhor caminho alternativo disponível através de qualquer sucessor de qualquer antecessor do nó atual. Se o nó atual exceder o limite, a recursão retorna para o caminho alternativo. À medida que retorna, o RBFS substitui o *f-value* de cada nó ao longo do caminho com o valor anterior (o melhor *f-value* do seu filho). Desta forma o RBFS lembra-se do *f-value* da melhor folha na subárvore esquecida e consegue decidir se vale a pena re-expandir a subárvore a alguma altura. RBFS é ótimo e a sua complexidade espacial é linear na profundidade da solução ótima mais profunda.
- SMA* funciona como o A*, expande a melhor folha até a memória ficar cheia. O algoritmo retira sempre o nó com a pior folha (o que tem maior *f-value*). A retornar o caminho para trás funciona como o RBFS. Com esta informação, o SMA* regenera a subárvore só quando todos os outros caminhos foram entendidos como sendo piores do que o caminho que o algoritmo esqueceu.

Hill-Climbing

Pseudo-código:

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

- O algoritmo de Hill climbing é um ciclo que se move continuamente na direção de valores crescentes, isto é, sobe (uphill). Termina quando atinge um pico onde nenhum vizinho tem um valor maior. Os algoritmos não mantêm uma árvore de procura, então a estrutura de dados para o nó corrente precisa só de guardar o estado e o valor da função objetivo. O algoritmo nunca analisa , num dado estado, para além dos seus vizinhos. O algoritmo guarda sempre um bom estado vizinho sem pensar sobre onde ir depois. O máximo local (local maxima) é um pico que é maior que cada um dos estados vizinhos mas mais baixo que o máximo global. Um cume (ridge) é uma sequência de máximos locais que é difícil para algoritmos greedy navegar. Planalto (plateaux) é uma área plana de um panorama espacial de estados. Pode ser um máximo local plano, para o qual não existe subida, ou um rebordo (shoulder) no qual o progresso é possível. Uma busca Hill-climbing pode perder-se no planalto. O algoritmo começa de forma randomizada e tenta chegar ao objetivo através daí. Existem outras variantes deste algoritmo, chamadas de Stochastic Hill climbing, First-choice Hill climbing e random-restart Hill climbing.

Simulated annealing

Pseudo-código:

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.VALUE – current.VALUE
    if ΔE > 0 then current ← next
    else current ← next only with probability e^(ΔE/T)
```

- Um algoritmo de Hill climb que nunca faz movimentos de descida com valores menores (ou custo maior) é garantido de ser incompleto, pois pode ficar preso num máximo local. Em contraste, mover-se para um sucessor escolhido de forma randomizada num conjunto de sucessores, é completo mas extremamente ineficiente. Por isso, parece razoável tentar combinar Hill climbing com movimentos random numa certa forma que mantêm ambos eficiência e completude. Simulated annealing é o algoritmo para isso. O algoritmo é muito semelhante ao de Hill climb, mas em vez de escolher o melhor movimento, escolhe um movimento randomizado. Se o movimento melhorar a situação, será sempre aceite, senão, o algoritmo aceita o movimento com uma probabilidade abaixo de 1. A probabilidade diminui exponencialmente com o quão pior o movimento é.

Beam Search

- Manter só um nó em memória poderá ser uma reação extrema para o problema de limitações de memória. O algoritmo local beam search mantêm rastreamento de um número *k* de estados em vez de só um. Começa com *k* estados randomizados, em cada passo, todos os sucessores de todos os *k* estados são gerados. Se algum for o objetivo, o algoritmo para, senão, seleciona os *k's* melhores sucessores da lista completa e repete o processo. Uma variante deste algoritmo, stochastic beam search, em vez de escolher o melhor *k* da lista de sucessores candidatos, escolher os *k* sucessores de forma random, com a probabilidade de escolher um dado sucessor a ser a função aumentada do seu valor. Esta variante tem alguma semelhança com o processo de seleção natural, onde os sucessores (*offspring*) se um estado (*organism*) forma a próxima geração de acordo com o seu valor (*fitness*).

Genetic Algorithms

Pseudo-código:

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

- Um algoritmo genético é uma variante da *stochastic beam search* no qual cada estado sucessor é gerado combinado dois estados pai em vez de modificar um estado único. A analogia de seleção natural é a mesma, exceto que agora lidamos com reprodução sexual em vez de assexual. Como a beam search, os algoritmos genéticos começas com um set de estados que foram formados de forma random, chamado de população. Cada estado, ou indivíduo, é representado sobre forma de uma string sobre um alfabeto finito (alfabeto que costuma ser composto por 0's e 1's). Cada estado é avaliado sobre uma função objetivo, a função de fitness. Uma função de fitness deve retornar valor mais altos para estados melhores. De notar que um indivíduo é selecionado duas vezes e outro nunca é selecionado. Para cada par ser acasalado, é criado um ponto de crossover das posições na string de forma random. Depois de feito o ponto de crossover, é realizada a mutação, que resulta no novo sucessor.

Minimax

Pseudo-código:

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

- O algoritmo Minimax calcula a decisão Minimax de um estado atual. Usa uma implementação recursiva para computar os valores Minimax de cada estado sucessor. A recursão procede até as folhas da árvore, e depois os valores de Minimax são retornados ao longo da árvore ao ritmo que a recursão volta ao início. O algoritmo faz uma busca em DFS da árvore de jogo.
- Complexidade espacial: O(bm), m é a profundidade máxima da árvore e b as ações legais.
- Complexidade temporal: $O(b^m)$.

Alfa Beta Pruning

Pseudo-código:

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

- O problema do Minimax é que o número de estados de jogo que tem de analisar é exponencial na profundidade da árvore. Infelizmente não podemos eliminar a exponencialidade, mas podemos cortar eficientemente em metade. A técnica de pruning (poda) faz isso, retorna os mesmo movimentos que o Minimax, mas elimina os ramos que não vão influenciar na decisão final. O corte alfa beta pode ser aplicado para árvores de qualquer profundidade, e é geralmente possível cortar subárvores inteiras para além de folhas. O princípio geral é o seguinte: considerar um nó *n* em algum lugar da árvore, posição tal em que o jogador tem a possibilidade de se mover para esse nó. Se o jogador tem uma melhor escolha *m* tanto no pai do nó *n* como em qualquer opção tomada a partir de *n* para a frente, então *n* nunca vai ser alcançado numa jogada. Então, quando sabemos o suficiente sobre *n* (analisando alguns dos seus descendentes) para tirar esta conclusão, podemos cortá-lo.
- Alfa: o valor da melhor (maior valor) escolha que encontramos até a um determinado momento ao longo do caminho de MAX
- Beta: o valor da melhor (menor valor) escolha que encontramos até a um determinado momento ao longo do caminho de MIN
- Complexidade espacial: O(bm), m é a profundidade máxima da árvore e b as ações legais.
- Complexidade temporal: $O(b^m)$.

Monte Carlo Tree Search (MCTS)

Pseudo-código:

MCTS Algorithm for Action Selection

```
repeat N times {  // N might be between 100 and 1,000,000
    // set up data structure to record line of play
    visited = new List<Node>()
    // select node to expand
    node = root
    visited.add(node)
    while (node is not a leaf) {
      node = select(node, node.children)  // e.g. UCT selection
      visited.add(node)
    }
    // add a new child to the tree
    newChild = expand(node)
    visited.add(newChild)
    value = rollOut(newChild)
    for (node : visited)
        // update the statistics of tree nodes traversed
        node.updateStats(value);
    }
}
return action that leads from root node to most valued child
```

- Para selecionar o melhor nó, o algoritmo usa o UCB para as árvores, sendo a fórmula:

$$UCB1 = \| Xj + C\sqrt{\frac{2\ln n}{nj}}$$

- Xj é a recompensa estimada da escolha j, n o número de vezes em que o pai foi visitado, nj o número de vezes em que a escolha j foi feita.
- Exploitation: 1ª Parcela da soma (esquerda):
    - Enfatiza a recompensa
    - Torna a busca mais guiada
- Exploration: 2ª Parcela da soma (direita):
    - Reforça a exploração dos nós menos frequentemente visitados
    - Reduz o efeito de “rollouts” com pouca sorte
    - Constante C equilibra Exploitation e Exploration.

Contraint Satisfaction Problems

- Os problemas de satisfação de restrições representam um estado com um conjunto de pares de variáveis / valores e representam as condições para uma solução por um conjunto de restrições sobre as variáveis. Muitos problemas importantes do mundo real podem ser descritos como CSPs.
- Várias técnicas de inferência usam as restrições para inferir quais pares de variável / valor são consistentes e quais não são. Estes incluem nó, arco, caminho e consistência-k.

A pesquisa de retrocesso, uma forma de pesquisa em profundidade, é comumente usada para solucionar os CSPs. A inferência pode ser entrelaçada com a pesquisa.

- Os valores mínimos restantes e as heurísticas de graus são métodos independentes de domínio para decidir qual variável escolher de seguida numa pesquisa de retrocesso. A heurística de valor de restrição mínima ajuda a decidir qual valor tentar primeiro para uma determinada variável. O backjumping dirigido por conflitos recua diretamente para a origem do problema.
- A busca local usando a heurística de conflitos min também foi aplicada para restringir os problemas de satisfação com grande sucesso.
- A complexidade de resolver um CSP está fortemente relacionada à estrutura do seu gráfico linear de restrição. Problemas estruturados em árvore podem ser resolvidos em tempo linear. O condicionamento de encapsulamento pode reduzir um CSP geral para um estruturado em árvore e é bastante eficiente se um pequeno recorte puder ser encontrado. As técnicas de decomposição de árvore transformam o CSP numa árvore de subproblemas e são eficientes se a largura da árvore do gráfico de restrições for pequena.