Contents

Search Graphs

- A *graph* consists of a set N of *nodes* and a set A of ordered pairs of nodes, called *arcs*
- Node n_2 is a *neighbour* of n_1 if there is an arc from n_1 to n_2
  - That is, if $\{n_1, n_2\} \in A$
- A *path* is a sequence of nodes $\{n_0, n_1, \ldots, n_k\}$ such that $\{n_{i-1}, n_i\} \in A$
- Given a set of *start nodes* and *goal nodes*, a *solution* is a path from a start node to a goal node

Graph Searching

- Generic search algorithm:
  - Given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes
- Maintain a *frontier* of paths from the start node that have been explored
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered
- The way in which the froniter is expanded defines the *search strategy*
- We assume that after the search algorithm returns an answer, it can be asked for more answers and the procedure continues

Depth-First Search

- *Depth first search* treats the frontier as a stack
- It always selects on the the last elements added to the frontier
- If the frontier is $\{p_1, p_2, \ldots, p_n\}$
  - p_1 is selected
    - Paths that extend $p_1 * areaddedtothefrontofthestack, infrontof$ p_{2}$
  - p_2 is selected when all the paths from p_1 have been explored

Complexity

- Depth-first search isn't guarenteed to halt on infinite graphs or on graphs with cycles
- The space complexity is linear in the size of the path being explored
- Search is unconstrained by the goal until it happens to stumble on the goal

Breadth-First Search

- *Breadth-first search* treats the frontier as a queue
- It always selects one of the earliest elements added to the frontier
- If the frontier is $\{p_1, p_2, \ldots, p_n\}$
  - p_1 is selected
    - Its neighbours are added to the end of the queue, after p_r
  - p_2 is selected next

Complexity

- The *branching factor* of a node is the number of its neighbours
- If the branching factor for all nodes is finite, breadth-first search is guaranteed to find a solution if one exists
- It is guaranteed to find the path with fewest arcs
- Time complexity is exponential in the path length: b^n where b is branching factor, n is path length
- The space complexity is exponential in path length: b^n
- Search is unconstained by the goal

When is it practical to use DFS vs BFS?

Lowest-Cost-First Search

- Sometimes there are *costs* associated with arcs
- The cost of a path is the sum of the cost of its arcs

$$\text{cost}(\{n_0, \ldots, n_k\}) = \sum_{i=1}^{k} \mid \{n_{i-1}, n_i\} \mid$$

- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost
- The frontier is a priority queue ordered by path cost
- It finds a least-cost path to a goal node
- When arc costs are equal $\Rightarrow$ breadth-first search

Heuristic Search

- Idea: don't ignore the goal when selecting paths
- Often there is extra knowledge that can be used to guide the search: *heuristics*
- $h(n)$ is an estimate of the cost of the shortest path from node n to a goal node
- $h(n)$ uses only readily obtainable information (that is easy to compute) about a node
- h can be extended to paths: $h(\{n_0, \ldots, n_k\}) = h(n_k)$
- $h(n)$ is an underestimate if there is no path from n to a goal that has path length less than $h(n)$

Best-First Search

- Select the path whose end is closest to a goal according to the heuristic function
- Best-first search selects a path on the frontier with minimal h-value
- It treats the frontier as a priority queue ordered by h

Complexity

- It uses space expontential in path length
- It isn't guaranteed to find a solution, even if one exists
- It doesn't always find the shortest path

Heuristic Depth-First Search

- It's a way to use heuristic knowledge in depth-first search
- Order the neighbours of a node (by h) before adding them to the front of the frontier
- It locally selects which subtree to develop, but still does depth-first search
- It explores all paths from the node at the head of the frontier before exploring paths from the next node
- Space is linear in path length
- It isn't guaranteed to find a solution
- It can get led up the garden path

A* Search

- A* search uses both path cost and heuristic values
- $\text{cost}(p)$ is the cost of the path p
- $h(p)$ estimates of the cost from the end of p to a goal
- Let $f(p) = \text{cost}(p) + h(p)$. $f(p)$ estimates of the total path cost of going from a start node to a goal via p

Algorithm

- A* is a mix of lowest-cost-first and best-first search
- It treats the frontier as a priority queue ordered by $f(n)$
- It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node

Admissibility

If there is a solution, A* always finds an optimal solution - first first path to a goal selected - if

- the branching factor is finite
- arc costs are bound above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ)
- $h(n)$ is an underestimate of the length of the shortest path from n to a goal node

Why is A* admissible?

- If a path p to a goal is selected from a frontier, can there be a shorter path to a goal?
- Suppose path p' is on the frontier
- Because p was chosen before p' and $h(p) = 0$
  - $\text{cost}(p) \leq \text{cost}(p') + h(p')$
- Because h is an underestimate
  - $\text{cost}(p') + h(p') \leq \text{cost}(p'')$ for any path p'' to a goal that extends p'
- So $\text{cost}(p) \leq \text{cost}(p'')$ for any other path p'' to a goal
- There is always an element of an optimal solution path on the frontier before a goal has been selected
- This is because, in the abstract search algorith, there is the inital part of every path to a goal
- A* halts, as the minimum g-value on the frontier keeps increasing, and will eventually exceed any finite number

Binary Decision Diagram

A Binary Decision Diagram (BDD) is a rooted, directed acyclic graph

- With one or two terminal nodes of out-degree zero labeled 0 or 1, and a set of variable nodes u of out-degree two
- The two outgoing edges are given by two functions $low(u)$ and $high(u)$, a variable $var(u)$ is associated with each variable node

Reduced Ordered Bindary Desicion Diagram

A BDD ir ordered is on all paths though the graph the variables respect a given linear order $x_1 < x_2 < \cdots < x_n$.

An OBDD is Reduced if

1. No two distrinct nodes u and v have the same variable name and low/high successor
2. No variable node u has identical low/high successor

Feasibility and Non-Determinism

Cobham's Theis: A problem is feasibly unsolvable iff some deterministic Turing machine (dTm) solves it in polynomial time

$$P = \{\text{problems a dTm solvers in polynomial time}\}$$

$$NP = \{\text{problem a non-deterministic Tm solves in polynomial time}\}$$

Clearly, $P \subseteq NP$. Whether $P = NP$ is the most celebrated open mathematical problem in computer science. $P \neq NP$ would mean non-determinism wrecks feasibility. $P = NP$ says non-determinism makes no different to feasibility.

Boolean Satifiability (SAT)

SAT: Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true/false to $x_1, \ldots, x_n$?

Checking that a particular assignment makes φ true is easy (P). Non-determinism (guessing the assignment) puts SAT in NP. But it SAT in P? There are 2^N assignments to try.

Cook-Levin Theorem: SAT is in P iff $P = NP$, e.g. $(x_1 \vee \bar{x_2} \vee x_3) \vee (\bar{x_1} \vee \bar{x_3})$

CSAT: φ is a conjunction of clauses, where a *clause* is an OR of literals, and a *literal* is a variable x_i or negated variable $\bar{x_i}$

k-SAT: every clause has exactly k literals

Constraint Satisfaction Problem (CSP)

- $CSP = \{V, D, C\}$
  - Variables: $V = \{V_1, \ldots, V_N\}$

- Domain: The set of d values that each variable can take, $D = \{R, G, B\}$
- Constraints: $C = \{C_1, \ldots, C_k\}$

- Each constraint consists of a tuple of variables and a list of values that the tuple is allowed to take for this problem
  - $[(V_2, V_2), \{(R, B), (R, G), (B, R), (B, G), (G, R), (G, B)\}]$
- Constraints are usually defined implicitly $\rightarrow$ A function is defined to test if a tuple of variables satisfies the constraint
  - Example: $V_i \neq V_j$ for every edge (i, j)