Graph Search

P@trik Haslum

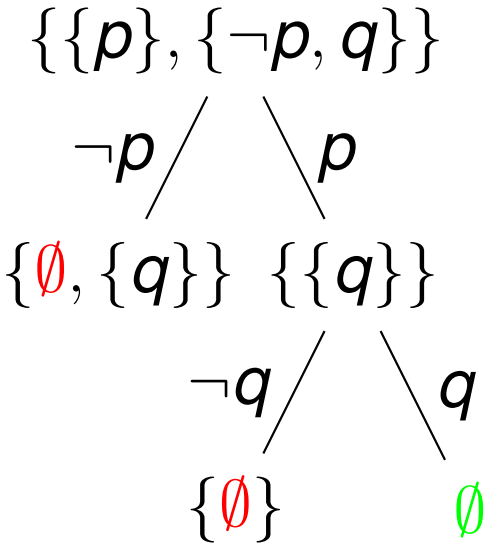
ANU/NICTA

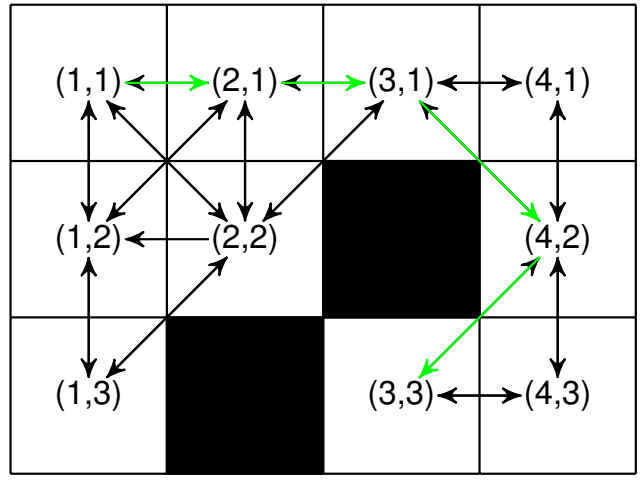
2012

Demo Code

http://users.cecs.anu.edu.au/~patrik/searchdemo/

“Tree” vs. “Graph” Search

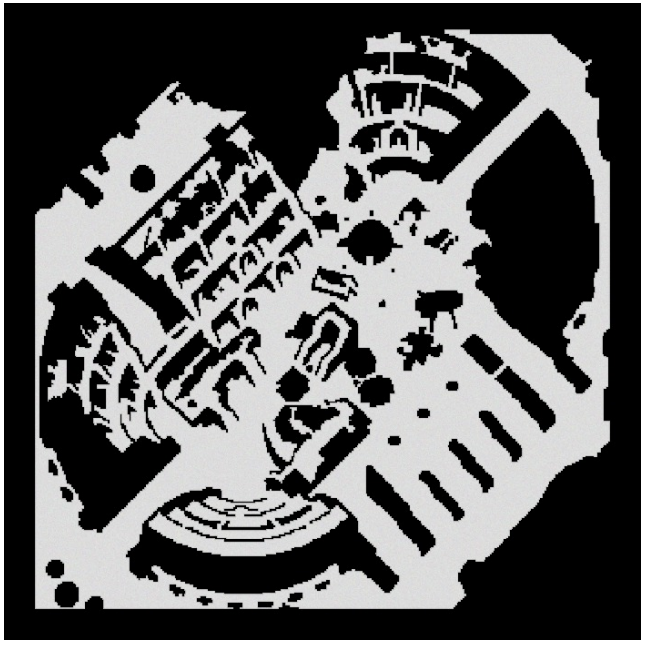


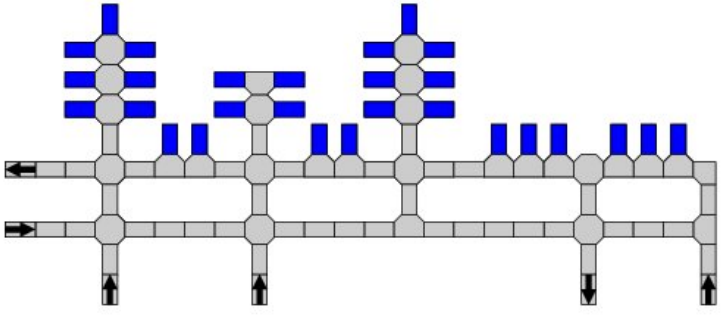


- Decision variables & constraints.
- Search space is a *tree*.
- Solution is a valid *node*.

- States & moves.
- Search space is a *graph*.
- Solution is a *path* from initial state to target state.

Single- and Multi-Agent Path-Finding



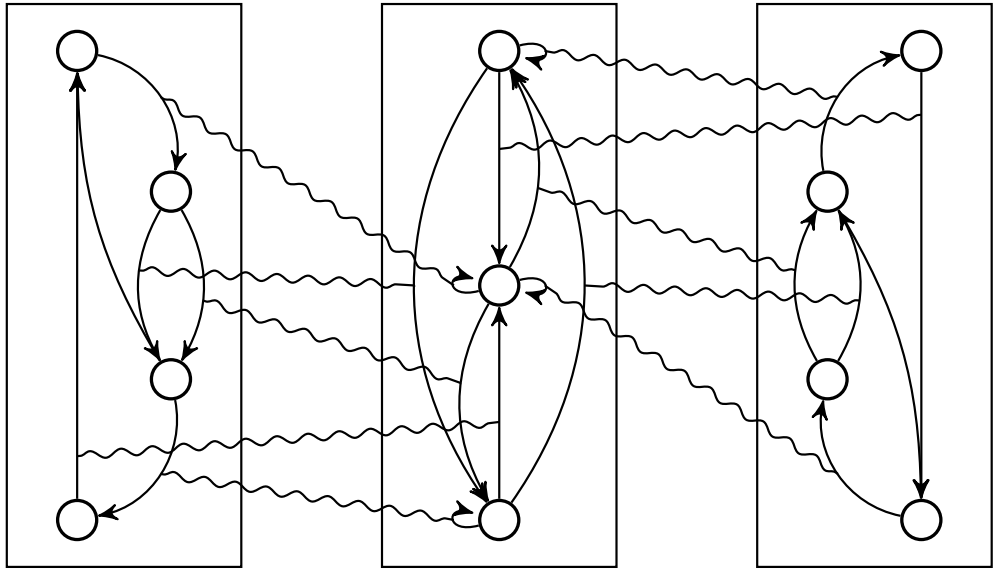


* Computer games, Robotics.
* Non-collision constraint.
* Movement constraints.
* Fixed discretisation (grid):
* Roadmap.
* k agents, n positions: $O(n^k)$ states.

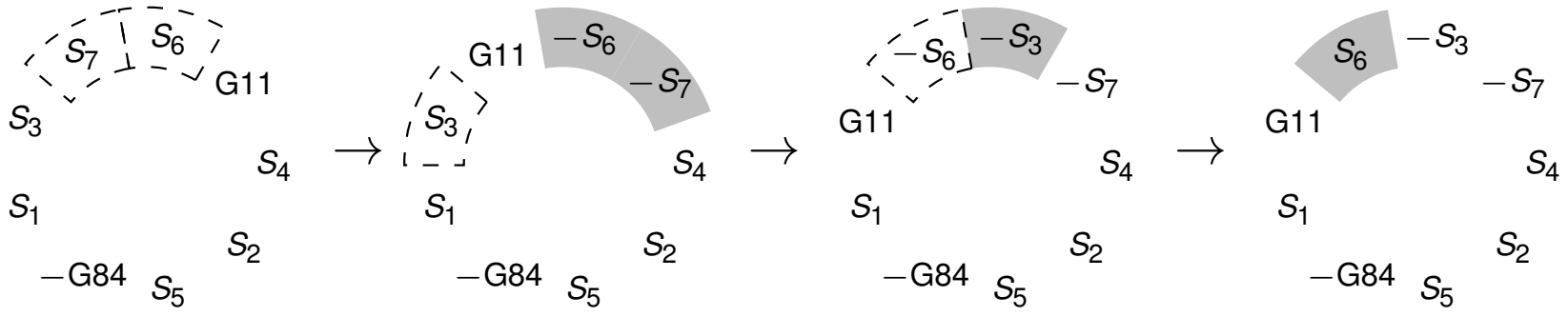
Reachability in Discrete State Systems

* Systems modelled by components with discrete states and (partially) synchronised transitions.
* k components with n states: $O(n^k)$ system states.

* Verification (model checking).
* DES Diagnosis.

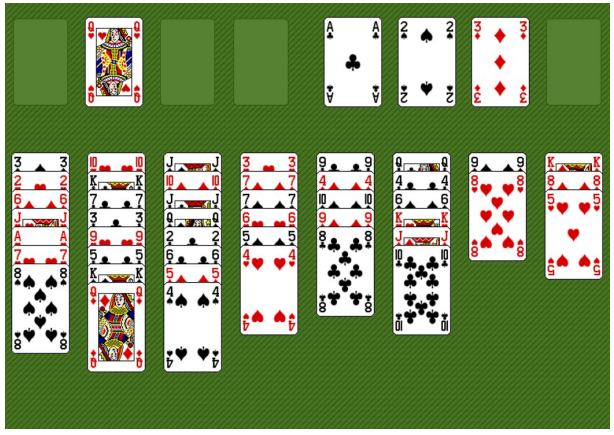


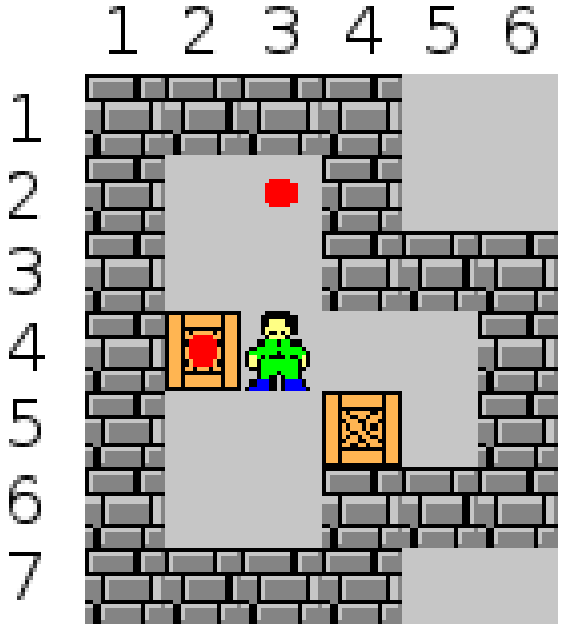
Genome Edit Distance Computation

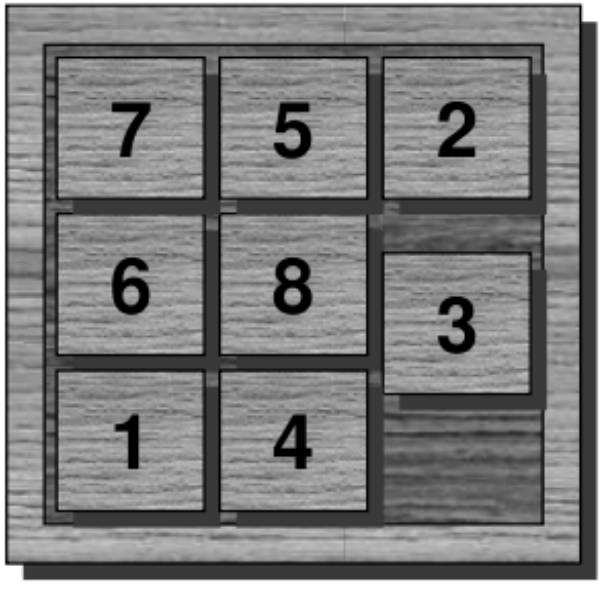


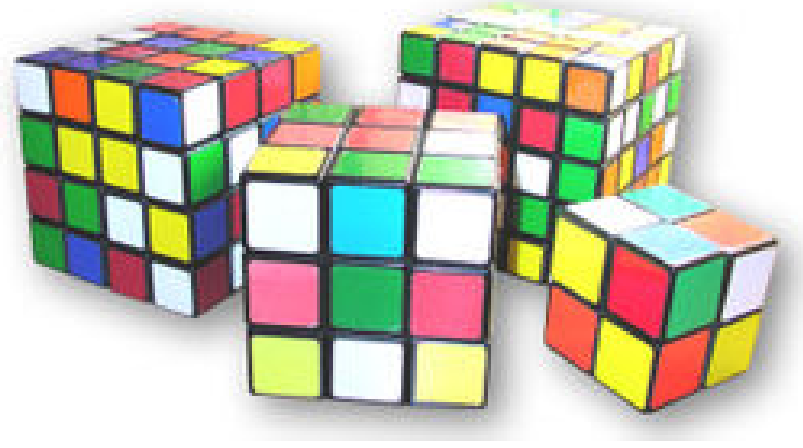
* States: Signed (cyclic) permutations.
  - n elements: $2^n \times (n-1)!$ states.
* Moves: Segment inversion, transposition & transversion.

Puzzles and Games









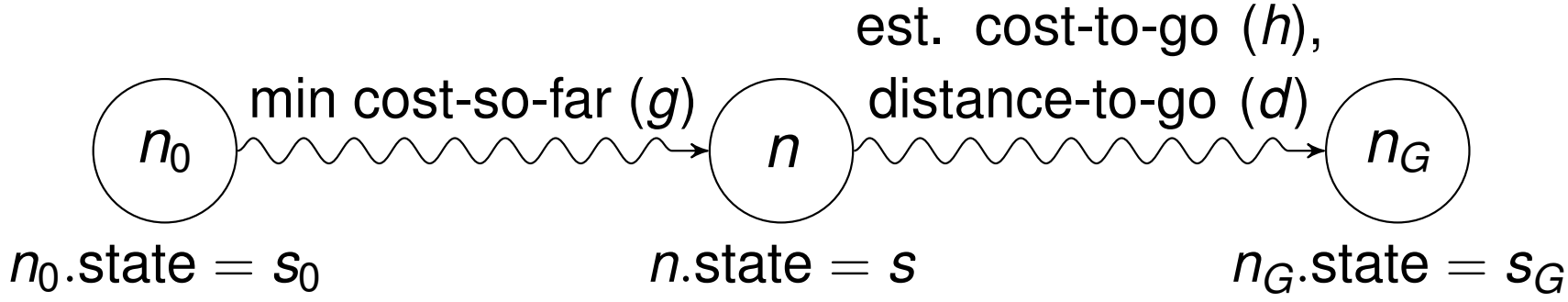
“Graph” vs. “Tree” Model

* A “graph” problem can be expressed in “tree” form:
  - Decision variables: 1^{st} move, 2^{nd} move, ...
  - Constraint: Sequence of moves forms a contiguous path in the state space.
* This requires a *bound* on path length:
  - If the bound is less than the length of the shortest optimal path, solution won’t be optimal (and if less than the length of the shortest path, we won’t find any solution).
  - Formulation grows linearly with bound.
  - Worst-case bound is the number of states.

Search Space Representation

* Explicit graph representation:
  - Ok for “small” graphs (e.g., roadmaps)
* Implicit (inductive) graph representation:
  - Initial state: s_0.
  - Successor function: $\text{succ}(s) = \{(m_1, s_1), \ldots, (m_k, s_k)\}$.
* Structured implicit representation:
  - States are assignments of values (from a finite domain) to (a fixed set of) *state variables*.
  - Moves are instances of *operators*, which have *preconditions* and *effects* on (a subset of) variables.
  - Typically exponentially compact.
* Minimise path cost $=$ sum of move costs along path.
  - No negative cycles, so minimum is well-defined.
  - Typically, assume $\text{cost}(m) \geq 0$ for all moves.

States and Nodes



- State: Element of the state space (graph vertex).
- Node: State reached from initial state via specific path.
- $g(n)$: Cost of cheapest path to n.state discovered so far.
- $h(s)$: Estimated cost of cheapest path from s to any goal state (*heuristic function*).
- $d(s)$: Estimated length of shortest path from s to goal.
- $g^\star(s)$: Cheapest cost of any path from s_0 to s.
- $h^\star(s)$: Cheapest cost of any path from s to any goal state.

Best-First Search

1: *open*, *closed* : Set of Node(state, g, parent).
2: Initialise *open* = $\{\text{Node}(s_0,0,\cdot)\}$, *closed* $= \emptyset$.
3: **while** *open* $\neq \emptyset$ **do**
4: **select best** node n in *open*
5: Remove n from *open*, add it to *closed*.
6: **if** n.state is a goal state **then**
7: **return** n.
8: **for** $(m, s') \in \text{succ}(n.\text{state})$ **do** // *expand n*
9: **if** $\nexists n' \in \textit{open} \cup \textit{closed} : n'.\text{state} = s'$ **then**
10: Add Node(s', $n.g + \text{cost}(m)$, n) to *open*.
11: **else if** $n.g + \text{cost}(m) < n'.g$ **then**
12: Set $n'.g = n.g + \text{cost}(m)$ and $n'.\text{parent} = n$.
13: **if** $n' \in \textit{closed}$ **then** // *Reopen n′*
14: Move n' back to *open*.
15: **return** *null*.

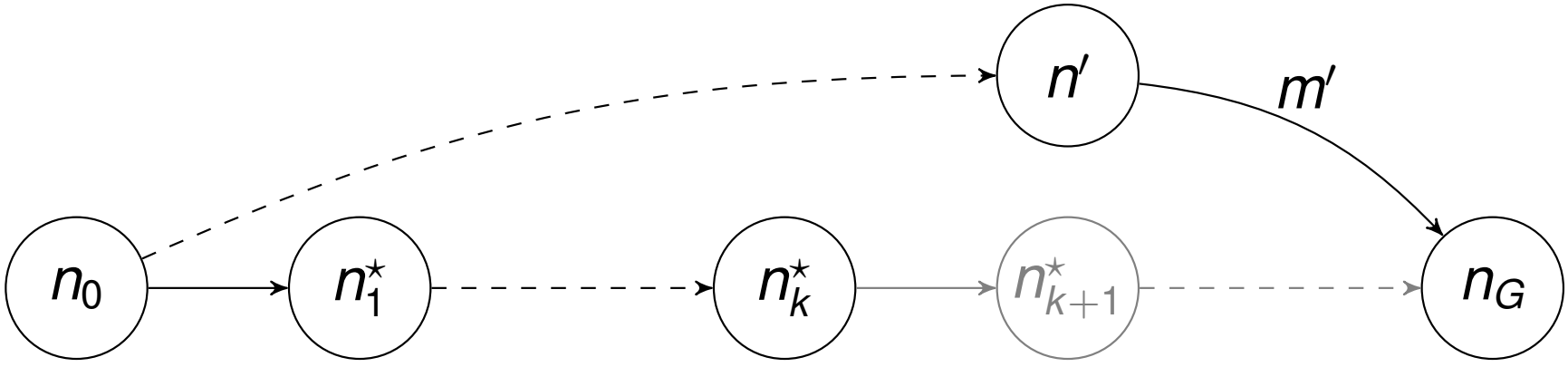
Which Node is “Best”?

* **Uniform-Cost Search**: n with min $n.g$.
* Counting cost(m) $= 1$ for all moves:
  - **Breadth-First Search**: n with min $n.g$.
  - **Depth-First Search**: n with max $n.g$.
* **Greedy Search**: n with min $h(n.\text{state})$.
* **A***: n with min $f(n) = g(n) + h(n.\text{state})$.

* Search is *informed* (a.k.a. *heuristic*) iff node evaluation uses estimated cost-to-go (h).
* Node reopening only required to ensure optimality – and not even always for that.
* If optimality not required, can return as soon as goal node is *generated*.

A*: Admissibility

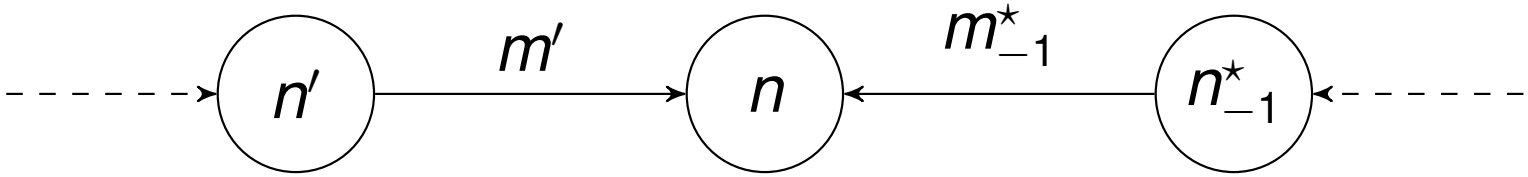
- h is *admissible* iff $h(s) \leq h^\star(s)$ for all s.
- **Theorem:** If h is admissible, A* returns an optimal path.



- At any time, some node on optimal path is in *open*.
- $f(n_{k+1}^\star) = g^\star(n_{k+1}^\star.\text{state}) + h(n_{k+1}^\star.\text{state}) \leq g^\star(n_{k+1}^\star.\text{state}) + h^\star(n_{k+1}^\star.\text{state}) = f^\star$.
- If $g(n') + \text{cost}(m') > f^\star$, $n_{k+1}^\star$ is selected before n_G with parent n'.

$A^\star$: Consistency

- h is *consistent* (a.k.a. *monotone*) iff $h(s) \leq \text{cost}(m) + h(s')$ for all s, $(m, s') \in \text{succ}(s)$.
- **Theorem:** If h is consistent, $A^\star$ reopens no node.
- To reopen n, must first close it (i.e., select it for expansion).



- Suppose $g(n') + \text{cost}(m') > g^\star(n.\text{state})$:
  $f(n^\star_{-1}) = g^\star(n^\star_{-1}.\text{state}) + h(n^\star_{-1}.\text{state}) \leq$
  $g^\star(n^\star_{-1}.\text{state}) + \text{cost}(m^\star_{-1}) + h(n.\text{state}) =$
  $g^\star(n.\text{state}) + h(n.\text{state}) < g(n') + \text{cost}(m') + h(n.\text{state})$
- $f(n') = g(n) + \text{cost}(m) + h(s') \geq g(n) + h(n.\text{state}) = f(n)$ for $(m, s') \in \text{succ}(n.\text{state})$ – f non-decreasing along path.
- Next open node on optimal path to n.state is selected before n with parent n'.

A*: Optimal Efficiency

- If h is admissible, any unexpanded node n can potentially lie on a path to goal with cost $f(n)$.
  - To prove optimality of path with cost $f^\star$, must expand every node with $f(n) < f^\star$.
- A* expands:
  - every node with $f(n) < f^\star$ – *once*, if h is consistent;
  - some nodes with $f(n) = f^\star$ (depends on tie-breaking);
  - and no node $f(n) > f^\star$.
- **Theorem:** No algorithm using the same consistent h, the same tie-breaking policy, and no additional information about the search space can guarantee path optimality with fewer node expansions than A*.

Can we do better?

* Use a better heuristic:
  - Heuristic inconsistency can lead to an exponential number of re-expansions – but only in really pathological cases.
  - A stronger inconsistent heuristic is typically better than a weaker consistent heuristic.
* Use tie-breaking:
  - Standard A* policy: Prefer node with smaller h.
  - $f_{\min} = (\min_{n \in open} f(n)) \leq f^\star$: If cheapest *generated* goal node has $g = f_{\min} + \min_m \text{cost}(m)$, it is optimal.
* Use more information about the search space:
  - Symmetry and partial-order reduction.
  - Geometric shortest-path pruning on roadmaps.

Greedy Search

* $d(n)$ estimates the number of moves to goal, which equals the number of nodes that must be expanded to reach the goal: following d only can reach the goal quicker.
* Greedy on h: similar, if path length and cost correlate.
* No *guarantee* on the number of nodes expanded.
* No *guarantee* on solution length or cost.

Weighted A* (WA*)

* BFS on $f(n) = (1 - \alpha)g(n) + \alpha h(n)$, where $\alpha \in [0, 1]$.
* Interpolates between A* and greedy:
  - $\alpha = 0$: Uniform-cost search.
  - $\alpha = 0.5$: A*.
  - $\alpha = 1$: Greedy search.
* Alternatively, $f(n) = g(n) + wh(n)$, where $w \geq 1$.
  - Equivalent to $\alpha = \frac{w}{w+1}$.
* **Theorem:** If h is admissible, the cost of the path returned by WA* is no more than $w = \frac{\alpha}{1-\alpha}$ times the optimal cost.

Continued Search

- Instead of returning the first goal node:
  - let n_{best} be the cheapest goal node discovered so far;
  - continue search until $\mathit{open} = \emptyset$ (or out of time/memory);
  - *prune* from *open* any n with $g(n) + h(n) \geq g_{\text{best}}$ (using admissible h).
- Any new solution will be strictly better than current n_{best}.
- With node reopening, continued search eventually finds an optimal solution (like branch-and-bound).
- Can use any evaluation function to select next node from *open*.

Do we really need *closed*?

* Storing closed nodes saves work but consumes memory.
  - Time (expansions) and space proportional to number of *states* (without reopening).
* Which closed nodes do we really need to store?
  - If the graph is acyclic, none.
  - If a goal state is reachable and the node evaluation function strictly increasing with path length, none.
  - Else, nodes on current path to the current node.
* **Depth-first search** can be seen as BFS, where
  - *open* is a stack (last in, first out); and
  - nodes with no open successor are removed from *closed*.
  - Time proportional to number of *paths*, space proportional to longest path length (*linear space*).

Recursive Depth-First Search

```
1: function DFS( n, stack )
2:   if n.state is a goal state then
3:     return n.
4:   for (m, s') ∈ succ(n.state) do
5:     if ∄n' ∈ stack : n'.state = s' then    // Cycle check
6:       Let r = DFS(Node(s', n.g + cost(m), n), [n, stack])
7:       if r ≠ null then
8:         return r
9:   return null.
```

```
1: DFS(Node(s_0, 0, ·), []).
```

Pruning and Backed-Up Values

* If h is admissible, $f_0(n) = g(n) + h(n.\text{state})$ is a lower bound on the cost of any path to goal through n.
  - Given a *cost bound* b, can prune any node with $f(n) > b$.
* If h is admissible,

$$f_k(n) = \min_{n' \in \text{succ}(n)} f_{k-1}(n')$$

  is also a lower bound on the cost of any path to goal through n.
  - $f_k(n)$ is f with a k-deep look-ahead: $f_k(n) \geq f_{k-1}(n)$.
  - Can use any cut-off (instead of fixed depth): $f_{(\cdot)}(n)$ is known as the *backed-up value* of n.

Recursive Search for an Optimal Path

- **Branch-and-Bound:**
  - Single continued DFS.
  - After first solution, set $b = g(n_{\text{best}})$ and prune nodes that cannot lead to a better solution ($f(n) \geq b$).
- **Iterative Deepening A* (IDA*):**
  - Repeated cost-bounded DFS with increasing bounds.
  - Initial bound: $b = h(s_0)$.
  - If no solution within current bound, set next bound to backed-up value of root.
- **Recursive Best-First Search (RBFS):**
  - DFS(n) with cost bound given by the best alternative node off the path to n.
  - “Simulates” A* by reconstructing path to best open node.

Recursive DFS Branch-and-Bound

1: **function** DFS(n, *stack*)
2: **if** n.state is a goal state **then**
3: Set $n_{\text{best}} = n$, $b = g(n_{\text{best}})$
4: **else**
5: **for** $(m, s') \in \text{succ}(n.\text{state})$ **do**
6: **if** $\nexists n' \in \mathit{stack} : n'.\text{state} = s'$ **then**
7: **if** $g(n) + \text{cost}(m) + h(s') < b$ **then**
8: DFS(Node($s', n.g + \text{cost}(m), n$), [$n$, *stack*])

1: Initialise $n_{\text{best}} = \mathit{null}$, $b = \infty$.
2: DFS(Node(s_0, 0, $\cdot$), []).

Recursive IDA* (DFS)

1: **function** DFS(n, *stack*, b)
2: **if** n.state is a goal state **then**
3: Set $n_{\text{best}} = n$.
4: **return** $g(n)$.
5: Set $f_{\text{min}} = \infty$. // backed-up value of n after the loop
6: **for** $(m, s') \in \text{succ}(n.\text{state})$ **do**
7: **if** $\nexists n' \in \mathit{stack} : n'.\text{state} = s'$ **then**
8: **if** $g(n) + \text{cost}(m) + h(s') \leq b$ **then**
9: $f_{\text{bu}} = \text{DFS}(\text{Node}(s', n.g + \text{cost}(m), n), [n, \mathit{stack}])$.
10: **if** $n_{\text{best}} \neq \mathit{null}$ **then**
11: **return** f_{min}.
12: $f_{\text{min}} = \min(f_{\text{bu}}, f_{\text{min}})$.
13: **else**
14: $f_{\text{min}} = \min(g(n) + \text{cost}(m) + h(s'), f_{\text{min}})$.
15: **return** f_{min}.

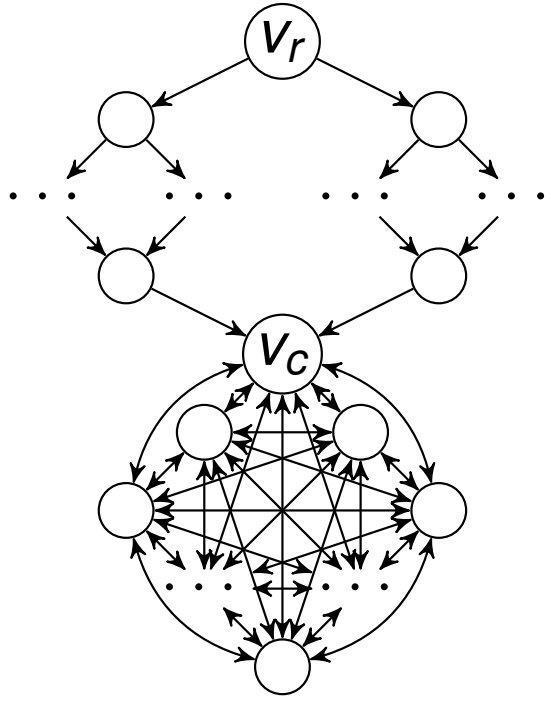
Recursive IDA* (Main)

1: Initialise $n_{\text{best}} = \mathit{null}$, $b = h(s_0)$.
2: **while** $n_{\text{best}} = \mathit{null} \wedge b < \infty$ **do**
3: $\quad b = \text{DFS}(\text{Node}(s_0, 0, \cdot), [], b)$.

* Why restart from the initial node?
  - Number of paths generally grows exponentially with depth:
  - DFS branch-and-bound performs unbounded DFS before discovering the first solution; can explore an arbitrarily deep (and thus large) subtree.
  - IDA* explores no path with cost $> f^\star$.
  - IDA*'s work is dominated by the last unsuccessful iteration.

A little memory can go a long way...

* If states are reachable by many paths, linear-space search (DFS/IDA*) performs much more work than BFS.
* A *transposition table* caches backed-up values of (a limited set of) states: use $T[s]$ in place of $h(s)$ if s encountered again.
* *However*, with cycle checking, the backed-up value depends on the *path*, not only state; naive CC + TT is incomplete and non-optimal.



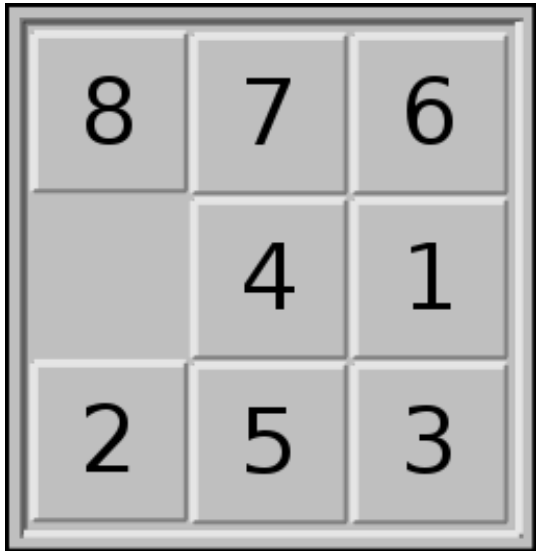
IDA*-GTT

- Instead of caching backed-up value, cache cost of cheapest path to s.
- In each iteration (i), need only expand s when reached by one cheapest path:
  - Prune n if $T[n.\text{state}] = (c, \cdot)$ and $g(n) > c$.
  - Prune n if $T[n.\text{state}] = (c, i)$ and $g(n) = c$.
  - Else, store $T[n.\text{state}] = (g(n), i)$ before expanding n.
- If no state on current path is ever dropped from T, this replaces the cycle check.

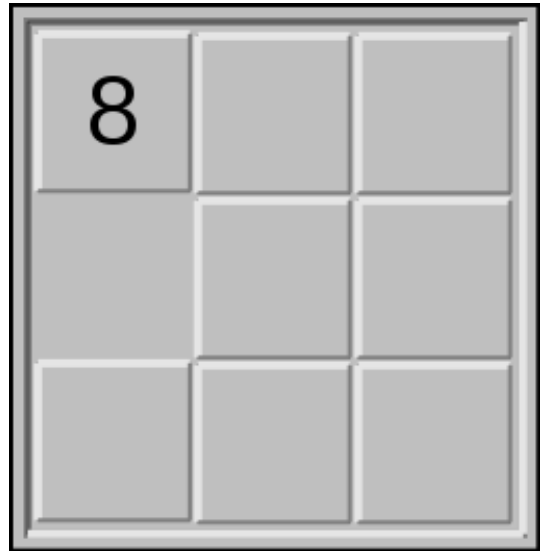
Where do Heuristics come from?

* The heuristic is (mostly) where “knowledge of the problem” enters search.
* Admissible heuristics:
  - Typically, optimal solution to a problem relaxation.
  - Manhattan Distance: $h(r, c) = |r - r_{\text{Goal}}| + |c - c_{\text{Goal}}|$.
    - Admissible, for 4-connected grid map.
    - Ignores obstacles and move costs.
* Combination of heuristics:
  - Maximum of admissible heuristics is admissible.
  - Sum yields a stronger heuristic, but only admissible if heuristics “count” cost of disjoint sets of moves.
  - Combining non-admissible heuristics: alternation.
* Memory-based vs. solving relaxation on-line.

Abstraction



181,440 states.



72 states.

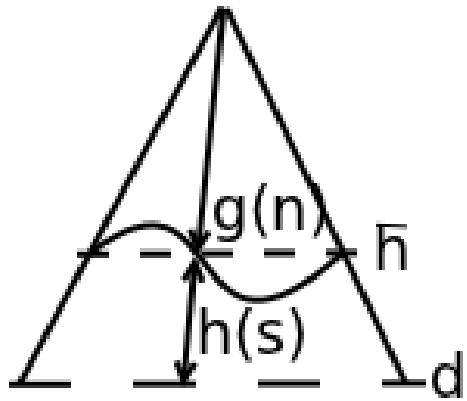
* Relaxation of structured problem representation.
* Optimal solution to abstract problem is an admissible heuristic for the original problem.
* Abstract space is small: Precompute optimal solution and store in a look-up table (“pattern database”, PDB).

Heuristic Impact on Search Efficiency

* Admissible heuristics in optimal search:
  - Generally, the higher heuristic values, the less search.
  - Difficult to quantify (without very strong assumptions).
* If h is *perfect* (i.e., $h = h^\star$), move costs are strictly positive and ties broken to lower h, A* expands exactly the nodes on one optimal path.
* If h has *constant error* (i.e., $h = \max(h^\star - C, 0)$, $C > 0$), $A^\star$ must expand all nodes on optimal paths up to $g = f^\star - C$.
  - It's easy to construct problems with an exponential number of states on optimal paths.
  - Assuming *no transpositions*, *a single goal state*, A* node expansions is exponential only in C.

* Consider a single, failed iteration of IDA* without memory (i.e., no transpositions);
* Assume uniform branching factor (b) and unit move costs.
* $E_h(c)$: # nodes expanded by a search to cost bound c.
* Assumptions imply $E_h(c) = |\{n \mid g(n) + h(s) \leq c\}|$.

* Blind search: $E_0(d) \approx b^d$.
* $E_h(d) \approx b^{d-\overline{h}}$.
  - $\overline{h}$: average heuristic value.
* But many more n with *high* $g(n)$ & *small* $h(s)$ are encountered in search.



* $E_h(d) \approx \sum_{k=0,\ldots,d} N_{d-k} P_h(k)$
  - N_i: # nodes with acc. cost (g-value) i.
  - $P_h(k)$: probability that $h(s) \leq k$, for s drawn uniformly at random from the search tree.

Constraints on Search

* Completeness: yes/no.
* Solution quality:
  - optimal – “reasonably good” – any solution at all.
* Time:
  - milliseconds – weeks.
* Space:
  - very limited (embedded device) – a few 100 Gb.
* Variation:
  - new search space every problem – only init/goal change.

Incomplete Search

* Local search:
  - Hill-Climbing, Simulated Annealing, Tabu Search, Random Walk.
  - Population-based: PSO, GA/EA, ACO, ...
  - (*Warning: gross over-simplifcation!*) May be good for optimising when feasibility is easy – i.e., when we can step from solution to solution.
* Beam Search, Iterative Broadening: BFS with limited *open*.
* BFS with aggressive pruning.

Any-Time and Bounded Suboptimal Search

* Bounded suboptimal search: Find any solution with cost no more than B times optimal.
  - WA*, A$^\star_\epsilon$, EES.
  - Use more information: admissible h, inadmissible h', distance estimate d.
* Cost-bounded search: Find any solution with cost $\leq C$.
* Any-time search: Find any solution quickly, and better solutions given more time.
  - Heuristic initial solution + branch-and-bound.
  - Iterated cost-bounded or bounded suboptimal search.

Large-Scale Complete/Optimal Search

* Invest effort in a very good heuristic, and strong admissible pruning.
  - Memory-based heuristics (PDBs).
  - Optimal additive combination of heuristics.
  - Symmetry breaking and partial-order reduction.
* Memory-limited search – but use all memory you have.
  - Linear-space search with transposition table.
  - SMA*.
* External memory search:
  - Store parts of *open*/*closed* not currently used on disk.
  - Requires organised disk access – can't rely on swap!
* Parallel search:
  - Exploit locality to avoid communication.
  - Very similar to disk-based search.

Repeated Search in the Same Space

* Graph stays the same, initial and goal state varies.
* Preprocess/simplify the problem:
  - Precompute and store admissible pruning conditions.
  - Identify useful short-cuts.
* Memory-based heuristics:
  - PDBs presume same goal (or intitial) state.
  - Transit heuristics.
* Precompute and store (compressed) all optimal solutions.