Solutions to exam(ple) questions AI

Joppe Geluykens

Search

Basic search

In the course, we studied depth-first, breadth-first, iterative deepening and bi-directional search. Discuss the properties of these methods: speed (worst time complexity), memory (worst space complexity) and completeness. Which technique would you choose, possibly depending on characteristics of the problem.

Completeness
: Depth-first: Yes for finite trees. Breadth-first: Yes, even finds shortest path. Iterative deepening: Yes, even finds shortest path. Bi-directional: Yes.

Speed
: Depth-first: $\mathcal{O}(b^d)$. Breadth-first: $\mathcal{O}(b^m)$ Iterative deepening: $\mathcal{O}(b^m)$. Bi-directional: $\mathcal{O}(b^{m/2})$

Memory
: Depth-first: $\mathcal{O}(b * d)$ Breadth-first: $\mathcal{O}(b^m)$ Iterative deepening: $\mathcal{O}(b * m)$ Bi-directional: $\mathcal{O}(b^{m/2})$

Heuristic search

In the course, we studied hill climbing, beam search, hill climbing 2 and greedy search. Briefly explain these methods. What are their properties in terms of speed (worst time complexity), memory (worst space complexity) and completeness.

Hill climbing

Depth-first, but expand child with best heuristic value first.

Completeness
: Yes, with backtracking.

Speed
: $\mathcal{O}(b^d)$

Memory
: $\mathcal{O}(b * d)$

Beam search

Breadth-first, but only keep $WIDTH$ best paths in queue.

Completeness
No.

Speed
$\mathcal{O}(WIDTH * b * m)$

Memory
$\mathcal{O}(WIDTH)$

Hill climbing 2

Beam search, but $WIDTH = 1$, so only best path in queue.

Completeness
No.

Speed
$\mathcal{O}(b * m)$

Memory
$\mathcal{O}(1)$

Greedy search

After each expanding of a node, sort the entire queue.

Completeness
Yes.

Speed
$\mathcal{O}(b^d)$

Memory
$\mathcal{O}(b^d)$

A* algorithm

Aspects of A*

Discuss the different aspects involved in the algorithm A*, more precisely:
Explain the uniform cost technique, What is wrong with uniform cost?
Explain the branch and bound idea, How can you integrate this idea in uniform cost?
How to integrate heuristics?
Why does this remain optimal for underestimating heuristics?
What is redundant path elimination?
How is this integrated in A*?
(4 or 5 lines and sometimes a drawing or illustration and normally sufficient for each ot these points)

1. **Uniform cost**: At each step, sort queue based on accumulated cost. Uniform cost is not always optimal. E.g.: last branch of the best path so far turns out to have a huge cost.

2. **Branch-and-bound**: don't stop until the first path in the queue has become the best path. (Continue uniform cost while first path does not reach goal.)

3. Instead of accumulated cost, **sort based on** $f(P) = h(endpoint_P) + cost(S..P)$. Remains **optimal for underestimating heuristics** because paths that have an actual cost higher than the best path will never become first in the queue.

4. **Redundant path deletion**: When a path to a node P has a higher cost than another path Q, containing P. Delete P from queue. A* compares end-nodes of new paths to other paths in the queue containing that node. If the cost of the new path is higher than the cost of an existing path containing the same node, remove the new path from queue.

Properties of A*

When is an A*-algorithm "more informed" than another A*-algorithm?
Illustrate the concept with an example.
Which result holds when an algorithm is more informed than another algorithm?
What is the practical relevance of that?
What is the monotonicity restriction? Illustrate with an example. Which result holds under monotonicity? What is the practical relevance of this result?
How can we ensure monotonicity for any A*-algorithm?

1. When it's **underestimating heuristic function returns a higher value** for the end-nodes of each path. The **practical relevance** is that an A* with a better heuristic function will most likely visit less nodes than an A* algorithm with a worse heuristic function, thus increasing speed.

2. A heuristic function satisfies the monotonicity restriction if for every arc A-B, $h(A) \leq h(B) + cost(A, B)$ and $h(G) = 0$. Practical relevance: each node is reached through the best path first. **Redundant path deletion** can be done more efficiently. When A* selects a path P, remember $end(P)$ has been selected. If later, a path Q is selected with $end(Q) = end(P)$, remove Q.

3. If we have an arc A-B and $f(S..AB) < f(S..A)$, correct the value of $f(S..AB)$ to be at least $f(S..A)$. **Redefine** f: $f'(S..AB) = max(h(B)+cost(S..AB), f'(S..A))$

Advanced search techniques

What problem with A* is the motivation for introducing IDA* and SMA*?

1. A* uses **extensive memory**, comparable to breadth-first search. In order to reduce memory usage, IDA* and SMA* are introduced.

IDA*

Explain how IDA* works. How does the f-bound change through the different iterations? Illustrate with some example (of your choice).
Discuss the properties of IDA* (memory, speed and optimality)?

1. IDA* is similar to iterative deepening search, in that it induces a **boundary** on the f-value of nodes that may be expanded (whereas iterative deepening sets a boundary on the depth). The initial f-bound = $f(S)$. Generate all children of S and set the new f-bound to be the minimum of all f-values $> f(S)$. Continue with that new f-bound instead of $f(S)$.

Optimal? Yes.

Speed? Depends on number of f-contours. Worst case, for every 2 paths $f(p) \neq f(q) : \mathcal{O}(N^2)$

Memory? Let σ be the minimal cost of an arc. $\mathcal{O}(b * \frac{cost(B)}{\sigma})$

SMA*

Explain the different new steps that SMA* adds to A*: what happens when the memory is full, sequential generation of children, giving up too long paths, propagation of f-values. When does it stop? What are the properties of SMA* (time, memory and optionality)?

1. When the memory is full, before expanding a new node, SMA* **removes the node with the highest cost** and remembers it's cost in the parent node.
2. Expand children 1 by 1, left to right.
3. When the length of a path will cover at least all the allocated memory, set the cost to **infinity**.
4. When all children M of a node N have been explored and for all M $f(S..M) > f(S..N)$, **adjust** $f(S..N) = min(f(S..M)|M child of N)$.
5. It stops when the cost of a path to a goal node is cheaper than any forgotten path.

Optimal? Yes, if memory is large enough to hold the optimal path. Otherwise, finds best path within memory constraints.

Speed? If enough memory to store entire tree, same as A*.

Memory? As much memory as allocated.

Machine Learning

Version Spaces

What is it used for? How do you handle a positive versus a negative example? How is VS initialised? When does it stop? Is your solution useful when it doesn't converge? Pros and cons?

1. To learn a certain concept based on a few examples. Other, unobserved examples, can then be judged as whether or not they will be part of the concept.
2. **Init**: Version Spaces G and S are initialized to be the largest and smallest hypotheses only.
   Positive: Add a minimal generalization to the specific hypotheses in S so that they do cover the positive example. Only consider generalizations of elements in S that are still more specific than some general hypothesis in G.

Prune all negative hypotheses that do not cover the positive example. Eliminate redundant hypotheses: remove hypotheses in S that are more general than other hypotheses in S.
Negative: Add a minimal specialization to the general hypotheses in G so that they do not cover the negative example. Only consider specializations of elements in G that are still more general than some specific hypothesis in S. Prune all positive hypotheses that do cover the negative example. Eliminate redundant hypotheses: remove hypotheses in G that are more specific than other hypotheses in G.

3. VS stops when it converged to a solution (G and S share the same hypothesis). Otherwise, VS stops because of no more examples. All hypotheses in G and S, and all intermediate hypotheses are still correct descriptions covering the test data. Else, if VS stops because S or G is empty, the test data was either inconsistent or the target concept could not be represented in the chosen hypothesis language.

4. An example can be classified positive if it is covered by all remaining hypotheses of the partially learned concept, negative it is not covered by any, or an estimate can be made if the example is covered by a few hypotheses.

5. **Pro**: positive and negative examples are dealt with in a completely dual way + VS does not need to remember previous examples. **Con**: VS cannot deal with noise. If an example is classified both positive and negative, the desired hypothesis is eliminated from the Version Space G.

Constraint processing

Backtracking

What are the 2 problems of backtracking? What info does the algorithm store and how is it used?

1. **Trashing**: If at a certain z_{depth} all assignments to z_{depth} fail, then the constraint on variables z_{depth} and z_x, $(x < depth$, x deepest constraint causing fail) fails for all possible values of $z_{depth-1}$ to z_{x+1}. Solution: backjump to z_x.

2. **Redundancy**: Checks on a certain constraint are done over and over again, computing the same values. Solution: backmarking (uses 2 arrays).

3. Backtracking stores the constraints and the assigned values to the variables. It checks the constraints with these variables and returns if it has an array of values (one for each level) that were successfully assigned.

Backjumping

How does backjumping solve the trashing issue of backtracking?

1. If it's known that all assignments for a certain z_{depth} fail on the constraint $c(z_x, z_{depth})$ (with x the deepest constraint causing the fail), then they will fail for all possible assignments to z_{x+1} until $z_{depth+1}$. The solution is to 'back-jump' and change the assignment of z_x.

Backmarking

How does backmarking solve the redundancy issue of backtracking?

1. It holds info in 2 arrays. The checkdepth array stores the deepest number of constraints that were checked for each combination $(z_k, a_{k,l})$ (with z_k the level and $a_{k,l}$ the value assigned to z_k at that level). The backup array holds the lowest level we backed up to between visiting 2 blocks for z_k.

2. **Properties**
   checkdepth$_{k,l} < k - 1$, last check was a fail.
   checkdepth$_{k,l} = k - 1$, last check can be either fail or success.
   checkdepth$_{k,l}$ < backup(k), $a_{k,l}$ caused fail previously and will cause fail again at the same depth.
   checkdepth$_{k,l}$ $\geq$ backup(k), all checks for variables z_m with m lower than backup(k) succeeded before and will succeed again.

Intelligent backtracking

Explain how no-goods and intelligent backtracking are related to backjumping and backmarking. Explain dynamic search rearrangement, applied to intelligent backtracking.

1. **Backjumping** when the reason for failure is due to a no-good at a higher level.

2. Each time a failing (no assignment possible due to no-goods) combination occurs again: don't check, backtrack (similar to **backmarking**).

3. Select the order of variables in the tree dynamically based on 'first fail principle'; if assigning a value to z_i is more likely to fail than assigning to z_j, then assign to z_i first. Practical relevance: smaller search tree if guess was right, less checks redone if guess only partially right.

Relaxation & Hybrid Constraint Processing - Waltz

Forward Check, Lookahead Check, AC1, AC3, Backtracking with Forward Check, Backtracking with Lookahead Check. Explain these techniques. Define 'constraint problem'. Which of the above techniques is the best choice for the Waltz procedure? Explain. Explain Waltz. Illustrate. How is it similar to constraint processing?

1. **Forward Check**: assign a value to z_i and check each constraint concerning z_i, reducing the domain of other variables.
2. **Lookahead Check**: activate each constraint exactly once.
3. **AC1**: Perform lookahead check, as long as elements are being removed. (= consistent)
4. **AC3**: Keep a queue of all constraints. Remove first element. Check the constraint on each variable involved. If domain of those variables is changed, add all other constraints involving those variables to the queue. Repeat until queue is empty. (= consistent)
5. **Backtracking with Forward Check**: Activate forward checking after each assignment to a variable.
6. **Backtracking with Lookahead Check**: Iniitially, perform a lookahead check. Activate lookahead checking after each assignment to a variable.
7. **Waltz**: Use AC3, but only constraints between junction piles of neighbouring junctions. Waltz keeps for each junction in a line drawing a pile of all possible labels. After expanding a new junction, constraints are checked on the neighbouring nodes and junction piles are reduced. Waltz stops when there are no conflicting labels anymore. It's similar to constraint processing, because certain labels cannot be neighbours, therefore reducing the junction piles based on those constraints.