

Search Revisited

2.1 Branch-and-Bound

The main problem with the basic search procedures is the exponential growth in size. We can make progress if we try to estimate the distance from a node to a goal. The first procedure, called *branch-and-bound*, finds not only the goal but the shortest path to it. It does this by retaining paths and extends the path with the shortest total distance.

BRANCH&BOUND

- initialize Priority Queue with only root-node path
- while (shortest path in Queue not to Goal) and (Queue not empty)
 - dequeue first path from Queue
 - append successors
 - sort Priority Queue by length of path
- if Goal reached announce success
else announce failure

Of course, it would be better to discard a path if one already has a shorter path to that node.

2.2 The A* Algorithm

For further improvement, we add an *estimate* of the distance from the end of the path to the goal. If f denotes the length of the current path and g is an estimate of the distance from the end of the current path to the goal, then overall estimate is

$$e(\text{total}) = f(\text{already}) + g(\text{remaining}).$$

We again explore the most promising first (lowest e -value).

If g is an over-estimate, then one might reject a path which needs exploration, and terminate with the path that is not the best. However,

if g is always an under-estimate, then we will never terminate until the best path is found.

For example, if $g \equiv 0$ then we have breadth first search.

We can make these ideas formal as follows. Any best-first search with evaluation function

$$e(n) = f(n) + g(n),$$

where $f(n)$ is the cost from start state to node n , and $g(n)$ is heuristic estimate of cost from n to goal, is known as an A algorithm. If $g(n)$ is a lower bound (an underestimate) for the cost of the shortest path from n to the goal, then it is called an **A*** *algorithm*.

A* ALGORITHM

- initialize Priority Queue with only root-node path
- while (shortest path in Queue not to Goal) and (Queue not empty)
 - dequeue first path from Queue
 - append successors
 - prune if longer path to same node
 - sort Priority Queue by estimate of length of entire path from Source to Goal
- if Goal reached announce success
else announce failure

An A* algorithm has some optimality properties. In particular it is **admissible**: it always find a minimum path (if such a path exists).

The two extremes are $g(n)$ the actual distance, and $g(n) = 0$. If $g_1(n) \geq g_2(n)$ for all n and both are underestimates, we say that g_1 is more informed than g_2 . The A* algorithm for g_1 will use at most the work of g_2 .

THE 15 PUZZLE. A common kids' game is a plastic device with 15 numbered square tiles and one open spot arranged in a 4-by-4 grid. The goal is to get from a jumbled state to the state where the squares are in order.

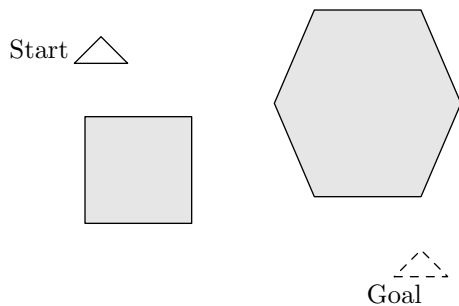
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Here a lower bound g on the distance can be obtained by adding up over all the tiles the number of moves each tile needs to make to end up in the right spot.

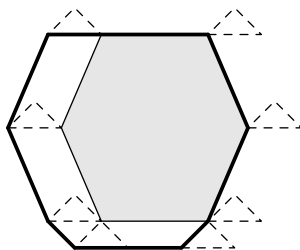
2.3 Robotic Planning: An Application

(Adapted from Winston:) Consider the problem of moving a robot to a goal position given some obstacles. The shape of the robot and the obstacles are known. To simplify the problem, we assume the robot is a triangle and is **not** allowed to rotate.

For example, consider the two shapes shown, with the target position dashed.

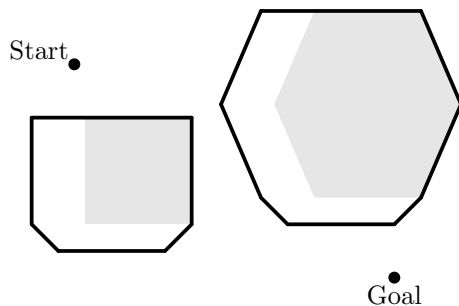


Then the solution is to focus on one corner C of the robot. For each obstacle, there is a forbidden area for the corner C . This can be visualized by sliding the robot around the obstacle (always touching) and seeing the curve traced out by the corner C .



If the obstacle is a polygon, then one can calculate this forbidden area by only considering those positions where some corner of the robot and some corner of the obstacle meet. Indeed, the forbidden curve is another polygon.

Then the question is whether the point C can reach the desired goal point while staying out of the forbidden areas. This is clearly a simpler problem.



Then, to use search to solve this problem, we construct a graph. The nodes of the graph are the corners of the forbidden areas as well as the start and goal points. Two nodes are joined iff one doesn't go through a polygon to get between them. Then the problem is to find the shortest path from the start node to the goal node in this graph—which is exactly what searching does.

GENERATE-AND-TEST

```
recurse(level) {  
  if (level is final)  
    then { if (solution) celebrate }  
  else  
    forall values of next variable:  
      if (okay) {  
        determine implications  
        recurse (level+1)  
      }  
}
```

The test for “okay” could be nothing or, for example in cryptarithmic, could entail checking that the value is different to previous values. But it is worth the effort to make the `okay` method reject as many values as possible. The net result is to interleave (depth-first) searching with exploitation of the constraints.

The other important programming choice is a heuristic to choose the next guess. A good idea is the most constrained letter. So one could choose the letter in the most dependencies; or one could choose the letter with the fewest remaining possibilities—the MRV (minimum remaining values) heuristic.

|| THE 8 QUEENS PROBLEM. *Consider the 8-by-8 board as used in chess. The goal is to place 8 queens on the board such that no two queens are in the same row, the same column or same diagonal.*

Here is a solution to the 4 queens problem.

	Q		
			Q
Q			
		Q	

For solving the 8 Queens problem by exhaustive search, it is simplest to place the queens row-by-row, starting at the first. Each time we place a queen, this precludes certain possibilities in the row below. A nifty way to keep track of this information is given in Knuth’s famous paper on “Dancing Links”.

Exploiting constraints

In some cases, the search is driven by the constraints themselves. So we might structure our program as follows:

CONSTRAINTSEARCH

- do propagate knowledge/dependencies
 until contradiction/solution/exhausted
- if (solution) then celebrate
 else if (contradiction) then backtrack
 else guess and recurse

In a specific problem, one of course needs to determine the rules of constraints. There are two types of constraints: *direct constraints* from the problem (e.g. the value is between 0 and 9), and *dependencies* between variables (e.g. that $(R+E)\%10=S$). One might do some preprocessing of the dependencies to enable rapid propagation.

One area of improvement is to change the type of search. In particular, try to backtrack to the “reason for failure”. For example, consider a timetabler which guesses a set of days, then determines a set of times, only to reach a conflict. If it turns out that one of the lecturers doesn’t work the day she is assigned, then it would make sense to go straight back to the day guess, rather than trying other times.

Dependency directed backtracking can help. For each constraint propagation (assumption), maintain the justification. When one reaches a contradiction, one determines a no-good set (minimal set of assumptions to be undone). From this set one picks an assumption to retract, and then propagates that retraction back through time. These ideas are used in *nonmonotonic reasoning* and *truth maintenance systems*.

Another approach is backtracking by “min conflict”. (In some sense a simplified version of simulated annealing—discussed below.) This procedure works very well for the 8 queens problem:

MIN-CONFLICT

repeatedly randomly correct worst conflict

The general constraint satisfaction problem can be abstractly modeled as a constraint graph, in which case it becomes a *graph coloring* problem. One can do this fast if the constraint graph “decomposes”.

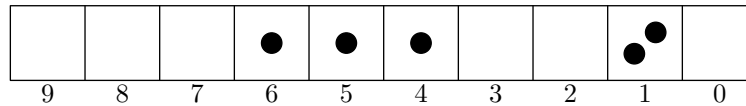
|| KAKURO. *Kakuro is like a crossword but with the digits 1 through 9 rather than letters. For each block the sum of the digits is specified, and within a block the same digit cannot be used twice.*

	7		14	27
11				
		5	16	
4				
	18			

An approach for computer search in Kakuro is to initially calculate the possible digits for each and every square. If we get down to a unique digit for a square, we ink it in and update. For example, consider the bottom square in the leftmost column. There is only one way to get 7 in 3 digits: 1, 2, & 4; there is only one way to get 4 in 2 digits: 1 & 3. Therefore, that square must be a 1.

Exercises

- 2.1. Solve the cryptarithmic given earlier.
- 2.2. The sliding-tile puzzle consists of three black tile, three white tiles and an empty space. These start in the configuration BBB~~e~~WWW. A tile may slide into the empty location; this has a cost of 1. A tile can hop over one or two other tiles into the empty position; this has a cost equal to the number of tiles jumped over. The goal is to have all the white tiles to the left of all the black tiles; the position of the blank is not important.
 - (a) Describe the state space for this problem and estimate its size.
 - (b) Propose a heuristic g for use in A*.
 - (c) Use A* to solve this problem.
- 2.3. In the “three-and-one” game, there is a 1-dimensional board, with squares numbered 0, 1, 2, etc as you go leftwards. Some counters are placed on the squares of the board. The object is to move all the counters onto square 0 in the fewest turns. The rules are that in one turn, one can
 - (a) move any **one** counter **three** squares to the right, or
 - (b) move **all** counters on a square **one** square to the right, or
 - (c) move **all** counters on a square **one** square to the left.



Write a program that provides a “good” player.

- 2.4. Consider the following sliding-block puzzle, where the object is to get the first row to read PANAMA and the second CANAL. (Recall that in each move, one can slide one tile adjacent to the blank space into the blank space.)

C	A	N	A	M	A
P	A	N	A	L	

Give an explicit lower bound heuristic g and sketch the first two steps of A* search on this problem.

- 2.5. **Project.** Code up a Kakuro solver.
- 2.6. An infinite sequence of positive integers is called a **broadcast sequence** if the distance between any two occurrences of the same number is more than that number: between two occurrences of i there must be at least i integers. For example, no two 1s can be consecutive. By optimal we mean that the maximum element is as small as possible. The optimal sequence when we are allowed to use 1 is 1, 2, 1, 3, 1, 2, 1, 3, ...

Use exhaustive search to try to find optimal broadcast sequences whose minimum elements are 2, 3 and 4 respectively.