Graph Explorations

Pierre-Edouard Portier

INSA Lyon

1. Some definitions about graphs

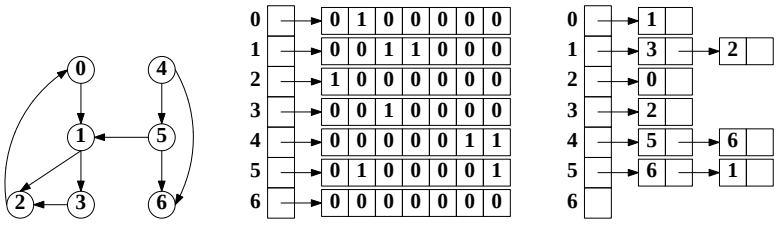
A *graph* $G = (V, E)$ is made of a set of *vertices* V and a set of *edges* E. An edge is a pair of vertices. This pair is ordered in case of *directed* graphs and unordered for *undirected* graphs. *Weights* can be assigned to the edges or the vertices. For example, in a road network the weight on an edge can represent the distance between two locations. A *path* is a sequence of connected edges. Two vertices are connected if there is a path between them.

An undirected graph is connected if every pair of vertices is connected. A directed graph is *weakly* connected if the underlying undirected graph is connected. A directed graph is *semiconnected* if, for every pair of vertices (u, v), there is either a directed path from u to v or a directed path from v to u. When for every pair of vertices there is a directed path in both directions, the directed graph is said to be *strongly* connected.

A graph is *acyclic* if there is no path linking a vertex to itself. A *tree* is a connected, acyclic and undirected graph.

A graph is defined explicitly when its whole structure is known prior to the traversal. A graph is said *implicit* when its structure is built during the traversal.

2. Graph data structures



Graph G1 **Adjacency matrix** **Adjacency lists**

A graph can be represented as an *adjacency matrix*. That is, an $n \times n$ matrix M where element $M[i, j]$ is the weight of edge (i, j) or 0 when there is no edge. The matrix explicitly stores the n^2 potential edges. When the number of edges is much less than n^2, the adjacency matrix is *sparse* (i.e., has many elements of value zero). If represented by a two dimensional array, it consumes unnecessary memory.

A more memory-efficient representation is made of *adjacency lists* where the neighbors of each vertex are stored in a linked list.

3. Generic graph traversal algorithm

Given an oriented graph$G$, what is the set of nodes that can be reached by starting from a node v and following the edges in all possible ways? Let s be the *successor* function that returns the direct neighbors of a set of nodes. For example, on graph$G1$:

$$s(\{1, 5\}) = \{6, 1, 2, 3\}$$

The *reachable* function can be defined as:

$$r(x) = x \cup r(s(x)) \tag{1}$$

For example, on graph $G1$, $r(\{1\}) = \{0, 1, 2, 3\}$.

When $r(x) = u$, there is also $r(u) = u$. In particular, it means that the successors of u are already contained in u.

$$r(u) = u \;\Leftarrow\; s(u) \subseteq u \tag{2}$$

The set u of nodes reachable from x is the smallest set that includes x and satisfies $r(u) = u$. It must be the smallest such set, otherwise it wouldn't be uniquely defined. For example, for all x, the set Ω of all nodes of the graph includes x and satisfies $r(\Omega) = \Omega$. It would also be necessary to prove that such a smallest solution exists. We could proceed by showing that for any two solutions u and v (i.e., $r(u) = u$ and $r(v) = v$), $u \cap v$ is also a solution. Thus, given such a set of nodes x, we could say that there is a smallest u^* such that $r(x) = u^*$ and $r(u^*) = u^*$ (viz. The intersection of all the sets u verifying $r(x) = u$ and $r(u) = u$).

Given the definition in equation (1), how to build a program that computes the set of nodes reachable from an initial node v? The postcondition to be attained by this program is:

$$r(\{v\}) = x \tag{R}$$

Postcondition (R) can be weakened into a loop invariant (P0).

$$r(\{v\}) = r(x) \tag{P0}$$

An invariant is a boolean expression that describes the values that can be taken by important variables. Initial values must be given to these variables so as to establish the invariant before entering the main loop of the program. The code within the loop is determined by the invariant since at the beginning of each new iteration, the invariant must be satisfied. Also, the invariant must be chosen such that its conjunction with the exit condition of the loop implies the postcondition.

In the case of (P0), the invariant is established before entering the loop by assigning $\{v\}$ to x. Within the loop, to maintain the invariant while progressing towards the postcondition, the following property is used:

$$r(x) = r(x \cup s(x))$$

Therefore, assignment $x \leftarrow x \cup s(x)$ maintains the invariant. Moreover, from equation (2) the exit condition appears clearly:

$$P0 \wedge (s(x) \subseteq x) \;\Rightarrow\; R$$

All the ingredients are there for a first version of the program.

```
x := {v}
WHILE ¬(s(x) ⊆ x)
    x := x ∪ s(x)
END WHILE
```

This version of the program can compute many times the same successors. To make it more efficient, a new invariant is introduced to distinguish nodes whose successors are already known (noted x and often called the *black nodes*) from those already visited but whose successors are unknown (noted y and often called the *grey nodes*). Not yet visited nodes are called the *white nodes*.

$$r(\{v\}) = r(x \cup y) \;\wedge\; x \cap y = \varnothing \;\wedge\; s(x) \subseteq (x \cup y) \tag{P1}$$

Establishing and maintaining this new invariant leads to an optimized version of the program.

```
x := ∅
y := {v}
WHILE y ≠ ∅
    x := x ∪ y
    y := s(y)\ x
END WHILE
```

It is still necessary to prove that this program terminates. The numbers of nodes not in *x* is at least equal to zero. At each iteration, this number is reduced by the size of *y*, a *strictly positive* value due to the entry condition of the loop. Therefore, the program must terminate.

Exercise Write a new version of this program so that function *s* is applied to individual nodes only. Try to find a solution that doesn't need to write statement `y := s(y)\ x` as a loop.

4. Breadth First Search (BFS)

When the grey nodes are represented by a *queue*, the generic graph traversal algorithm is known as *breadth first search*.

Exercise Study the implementation provided in file `bfs.cpp` and adapt it to compute the distances (number of edges in the shortest path) from a given source to all the reachable nodes.

5. Depth First Search (DFS)

When the grey nodes are represented by a *stack*, the generic graph traversal algorithm is known as *depth first search*.

Exercise Study the implementation provided in file `dfs.cpp` where the stack is implemented implicitly by recursion. Write a new version of the program with an explicit stack.

6. Dijkstra shortest path algorithm

Consider a graph where each edge is associated with a positive number, its weight. What are the shortest distances from a given node *v* to all the nodes reachable from *v*?

Let the *black nodes* be the ones for which the shortest distance to *v* is already established. On a shortest path, all the nodes are black.

Let the *grey nodes* be the ones for which a path from *v* has been found, but it may not be the absolute shortest one. However, it is the shortest one among the paths that are only made of black nodes except for this last grey node.

All paths from *v* to a *white node* must contain a grey node.

Giving meaning to the grey nodes and the black nodes sets the invariant at the core of Dijkstra's shortest path algorithm. The purpose is to design the main loop so that the invariant is maintained and the number of black nodes increases.

Let $d[u]$ be the length of the shortest path currently discovered from *v* to *u*. Such a path is necessarily made of black nodes. When *u* is a black node, $d[u]$ is the shortest distance separating *v* from *u*.

The following figure will support the argument.

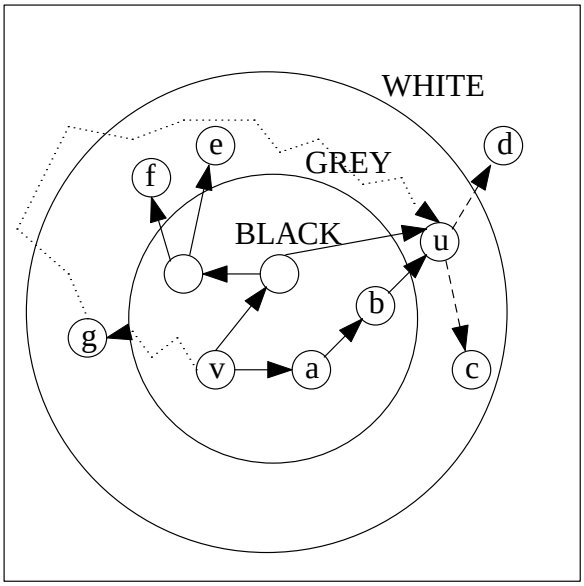


Illustration of Dijkstra's Shortest Path Algorithm

Let $w(x, y)$ be the positive weight associated with edge (x, y). On the above example, let $d[u]$ be $w(v, a) + w(a, b) + w(b, u)$.

Let u be the grey node minimizing $d[u]$. Consider any path from v to u. Let$l$ denote its length. It can contain grey nodes or even white nodes. v being black and u being grey, this path must leave the black region. Let g be the first non-black node (i.e. grey). Since u is a grey node chosen to minimize $d[u]$, $d[u] \leq d[g]$. Since, the distance between g and u is positive, $d[g] \leq l$. Therefore, by transitivity, $d[u] \leq l$. Thus, any path from v to u must be at least as long as the one of length $d[u]$. That is why u can join the black nodes.

If u becomes a black node, changes must be made to maintain the invariant. Each white successor$d$ of u becomes grey and $d[d] = d[u] + w(u, d)$. Each grey successor c of u stays grey, but the current $d[c]$ can be updated if it is greater than $d[u] + w(u, c)$.

```
x := ∅
y := {v}
WHILE y ≠ ∅
    // choose a grey node u for which d[u] is minimal
    x := x ∪ {u}
    y := y \ {u}
    FOR n ∈ s(u)
        CASE   n∈ x   → do nothing
             | n ∈ y   → d[n] := min(d[n], d[u] + w(u,n))
             | n ∉ x∪y → y := y ∪ {n} ; d[n] := d[u] + w(u,n)
        END CASE
    END FOR
END WHILE
```

Exercise Study the implementation provided in file `dijkstra.cpp` where the grey nodes are represented by a priority queue. What is the relationship between Dijkstra's algorithm and bfs or dfs?

7. Heuristics

In the context of finding a shortest path in a graph, let h be a function that associates to each node an estimation of its distance to the target. It is called a *heuristic* function.

Let $\delta(u, t)$ be the distance between u and t, i.e. The length of a shortest path. h is *admissible* if for each node u, $h(u) \leq \delta(u, t)$.

Heuristic h is *consistent* if for all edges (u, v), $h(u) \leq h(v) + w(u, v)$.

Let $(u_{0,\ldots,} u_k)$ be a path and $g(u_i)$ be the length of the path $(u_{0,\ldots,} u_i)$. Let f be a function defined by $f(u_i) = g(u_i) + h(u_i)$. Heuristic h is *monotone* if for all i and j such that $0 \leq i < j \leq k$, $f(u_i) \leq f(u_j)$. It means that the estimation of the optimal length of a path doesn't decrease with the addition of a node.

Consistency and monotonicity are equivalent properties. For two adjacent nodes u_{i-1} and u_i, we prove that consistency implies monotonicity:

$$
\begin{aligned}
& f(u_i) \\
= & \{Def.\ f\} \\
& g(u_i) + h(u_i) \\
= & \{Def.\ g\} \\
& g(u_{i-1}) + w(u_{i-1}, u_i) + h(u_i) \\
\geq & \{\ h \text{ is consistent } \} \\
& g(u_{i-1}) + h(u_{i-1}) \\
= & \{Def.\ f\} \\
& f(u_{i-1})
\end{aligned}
$$

We also have to prove that monotonicity implies consistency:

$$
\begin{aligned}
& h \text{ is monotone} \\
\Rightarrow & \{Def.\ of\ monotonicity\} \\
& f(u_i) \geq f(u_{i-1}) \\
= & \{Def.\ f\} \\
& g(u_i) + h(u_i) \geq g(u_{i-1}) + h(u_{i-1}) \\
= & \{Def.\ g\} \\
& h(u_{i-1}) \leq h(u_i) + w(u_{i-1}, u_i) \\
= & \{Def.\ of\ consistency\} \\
& h \text{ is consistent}
\end{aligned}
$$

It is also important to note that a consistent heuristic is admissible. For any path p from node u to target t, $p = (v_0 = u, \ldots, v_k = t)$, we have:

$$
\begin{aligned}
& w(p) \\
= & \{\textit{Def. of the weight of a path}\} \\
& \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \\
\geq & \{h \textit{ is consistent}\} \\
& \sum_{i=0}^{k-1} h(v_i) - h(v_{i+1}) \\
= & \{\textit{arithmetic}\} \\
& h(u) - h(t) \\
= & \{h(t) = 0\} \\
& h(u)
\end{aligned}
$$

Since the argument is made for any path p, we have in particular that $h(u) \leq \delta(u, t)$.

8. A* algorithm

To each node u of the graph, associate $f(u) = g(u) + h(u)$ where $g(u)$ is the length of the shortest path currently discovered between to join u from the source, and $h(u)$ is a consistent heuristic. Dijkstra's shortest path algorithm associates to each node u the value $g(u)$. What happens if $f(u)$ is used instead of $g(u)$? Let consider a grey node u minimizing $f(u)$. Let consider a direct successor v of u. When u becomes a black node, the current value $f(v)$ should be updated if it is smaller than:

$$
\begin{aligned}
& f(v) \\
= & \{\textit{Def. } f\} \\
& g(v) + h(v) \\
= & \{\textit{we consider joining } v \textit{ when coming from } u\} \\
& g(u) + w(u, v) + h(v) \\
= & \{\textit{arithmetic}\} \\
& g(u) + h(u) + w(u, v) + h(v) - h(u) \\
= & \{\textit{Def. } f\} \\
& f(u) + w(u, v) + h(v) - h(u)
\end{aligned}
$$

Therefore, taking into account heuristic h to find a shortest path to the target is equivalent to applying Dijkstra's algorithm on a reweighed graph where:

$$\hat{w}(u, v) = w(u, v) + h(v) - h(u)$$

The heuristic must be consistent for the weights to remain positive and for Dijkstra's algorithm to be correct. This algorithm is usually named A*. Let $f^* = \delta(s, t)$ be the length of the optimal solution from the source to the target. Based on Dijkstra's algorithm, A* visits all the nodes u for which $\delta(s, u) < f^*$. In fact, no algorithm could find with certainty the optimal solution without visiting at least as many nodes as A*.

Given two admissible heuristics h_1 and h_2, when, for each node u, $h_1(u) \leq h_2(u) \leq \delta(u, t)$, h_2 is said to be *more informed* than h_1.

Exercise Study the implementation provided in file `astar.cpp` where the algorithm is applied to the *n-puzzle** problem. Compare the number of visited nodes for different heuristics. How much memory does this algorithm consume? Is it a strong limitation?

9. IDA* algorithm

The *iterative deepening* strategy is able to consume as little memory as dfs while finding the shortest path to the solution like bfs does. It consists in a succession of independent bounded dfs where the maximal depth is increment by one at each iteration. The algorithm terminates when the solution is found.

```
PROC IDDFS(s, g) // s: starting node, g: goal node
    maxDepth := 0
    path     := () // the path to the solution is initially an empty list
    WHILE path = ()
        path := BDFS(s, g, 0, maxDepth)
        maxDepth++
    END WHILE
END PROC

PROC BDFS(n, g, depth, maxDepth) // DFS to explore no deeper than maxDepth
    IF n = g             THEN RETURN (n) END IF
    IF depth >= maxDepth THEN RETURN ( ) END IF
    FOR m ∈ s(n)
        path := BDFS(m, g, depth+1, maxDepth)
        IF path ≠ () THEN RETURN (n) ++ path END IF
    END FOR
    RETURN ()
END PROC
```

Given the bfs-ordering of a graph, most of the nodes will usually be found at the bottom level. Indeed, when the average number of successor of a node (also called the *branching factor*) is strictly greater than 1, the number of nodes at each successive level of a bfs traversal increases exponentially. Therefore, iterative deepening can save a lot of memory while having a moderate impact on the number of operations.

* https://en.wikipedia.org/wiki/15_puzzle

This same strategy can be applied to the A* algorithm to give IDA*.

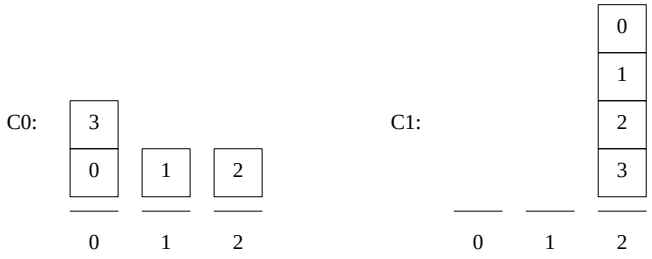
```
PROC IDA(startState)                // next upper bound initialized to the
    nextUpperBound := h(startState) // heuristic value of startState
    bestPath := ()
    path := (startState)
    WHILE bestPath = () ∧ nextUpperBound ≠ ∞
        upperBound := nextUpperBound
        nextUpperBound := ∞
        // nextUpperBound will be updated by SEARCH to the minimum
        // of all the depths that exceeded the current upperBound
        SEARCH(startState, 0, upperBound, nextUpperBound, path, bestPath)
    END WHILE
    RETURN bestPath
END PROC

PROC SEARCH(currentState, g, upperBound, nextUpperBound, path, bestPath)
    // g is the length of the shortest path currently discovered
    // to join currentState
    IF currentState = goal
        // the path to the goal must be stored in a fresh variable
        // since the variable "path" will be emptied while climbing up
        // the recursive ladder
        bestPath := path
        RETURN
    END IF                          // neighbors already on path are
    FOR n ∈ s(currentState) \ path  // excluded
        // f: underestimation of the distance to the goal
        f := g + w(currentState, n) + h(n)
        IF f > upperBound
            IF f < nextUpperBound
                nextUpperBound := f
            END IF
        ELSE
            path.push_back(n)
            SEARCH(n, g + w(currentState, n), upperBound, nextUpperBound,
                   path, bestPath)
            path.pop_back()
            IF bestPath ≠ () THEN RETURN END IF // solution found
        END IF
    END FOR
    RETURN // solution not found
END PROC
```

Exercise Study the implementation provided in file `ida.cpp` where the algorithm is applied to the n-puzzle. Note that the configurations in the neighborhood of the current state are not anymore stored explicitly, only the difference between two configurations is stored. What is the effect of sorting the neighbors by order of their heuristic value?

9.1. Project: blocks world

The purpose of this project is to solve the famous* *Blocks world* planning problem. A configuration is made of *n blocks* and *k stacks*.



Configurations C0 and C1 both have 4 blocks and 3 stacks. At each step, a block on the top of a stack can be moved to the top of another stack. The goal is to discover the shortest sequence of moves to go from one configuration to another. In our experiments, an initial configuration is made of the blocks sequentially distributed on the stacks from left to right and from bottom to top (see for example C0 above). Also, a final configuration is made of all the blocks on the last stack in ascending order from top to bottom (see for example C1 above).

Exercise Write an IDA* solver for the blocks world problem. You can use the state representation in file `blocks-world-state.cpp`. File `blocks-world-state-example.cpp` gives an example of how to use the class State. You should design your own heuristic functions to find the optimal solution for problems such as $n = 20$ blocks and $k = 4$ stacks.

10. ε-optimality and weighted-A*

A search algorithm is ε-optimal if it terminates by finding a solution of cost at most equal to $(1+\varepsilon)\ \delta(s,t)$ with s, t and ε standing for, respectively, the initial configuration, the target and a positive constant. For an admissible heuristic h, A* is ε-optimal when using the cost function $f'(u) = g(u) + (1+\varepsilon)\ h(u)$. Indeed, for u a minimal grey node, there is:

$$
\begin{aligned}
& f'(u) \\
=\ & \{\textit{Def. f'}\} \\
& g(u) + (1+\varepsilon)\ h(u) \\
=\ & \{\textit{u is a minimal grey node, Dijkstra's invariant}\} \\
& \delta(s,u) + (1+\varepsilon)\ h(u) \\
\leq\ & \{\textit{h is admissible}\} \\
& \delta(s,u) + \delta(u,t) + \varepsilon\ \delta(u,t) \\
=\ & \{\textit{For a shortest path from s to t through u}\} \\
& \delta(s,t) + \varepsilon\ \delta(u,t) \\
\leq\ & \{\textit{For a shortest path from s to t through u}\} \\
& \delta(s,t) + \varepsilon\ \delta(s,t) \\
=\ & (1+\varepsilon)\ \delta(s,t)
\end{aligned}
$$

When the target is discovered, $f'(t) \leq (1+\varepsilon)\ \delta(s,t)$. Weighted-A* is a non-optimal algorithm based on the cost function $f(u) = g(u) + \omega\, h(u)$, with ω a positive constant. See `wida.cpp`.

* https://en.wikipedia.org/wiki/Blocks_world