Motion Planning | **Graph Search II**

Autonomous Mobile Robots

Martin Rufli – IBM Research GmbH

Margarita Chli, Paul Furgale, Marco Hutter, Davide Scaramuzza, Roland Siegwart

Deterministic graph search | overview

- Encompasses deterministic optimization algorithms operating on graph structures $G(N, E)$
- The methods find a (globally lowest-cost) connection between a pair of nodes

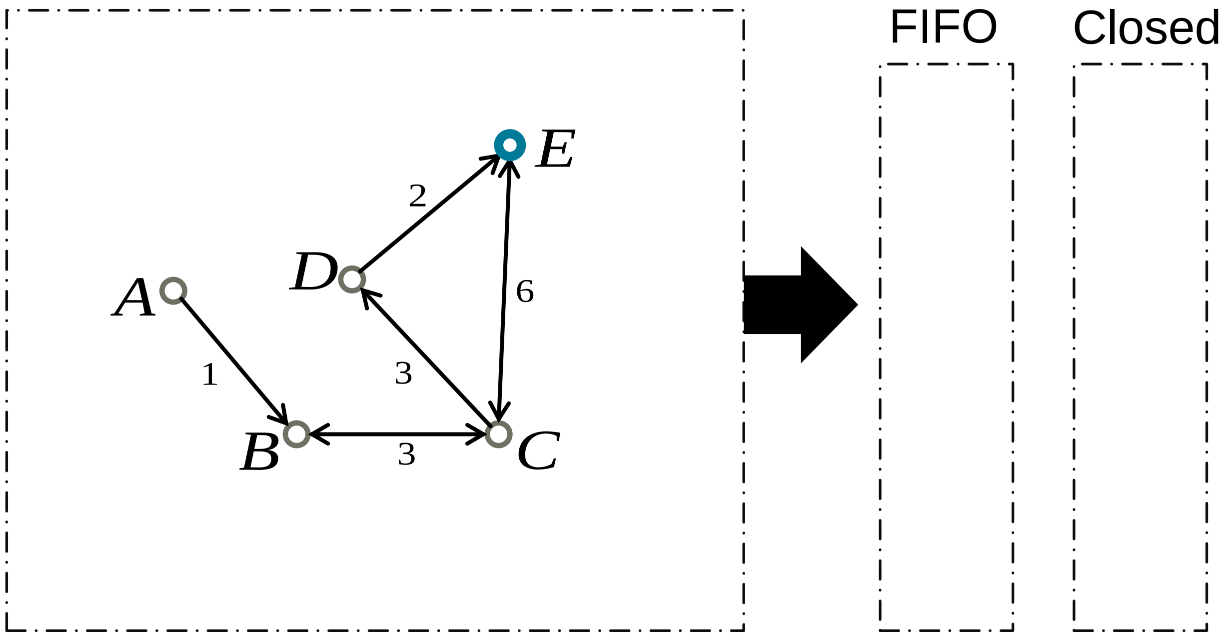
Breadth-first search | working principle

- The method expands nodes according to a FIFO queue and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

Breadth-first search | working principle

- The method expands nodes according to a FIFO queue and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

```
BF(Graph G, Node Start, Node Goal)
    Queue.init(FIFO)
    Queue.push(Start)
    while Queue is not empty:
        Node curr = Queue.pop()
        if curr is Goal return
        Closed.push(curr)
        Nodes next = expand(curr)
        for all next not in Closed:
            Queue.push(next)
```



Breadth-first search | properties

- The trajectory to the first goal state encountered is optimal iff all edge costs on the graph are identical and positive
- Optimality of the solution is retained for arbitrary positive edge costs, if search is continued until queue is empty
- Breadth-first search has a time complexity of $O(|V| + |E|)$

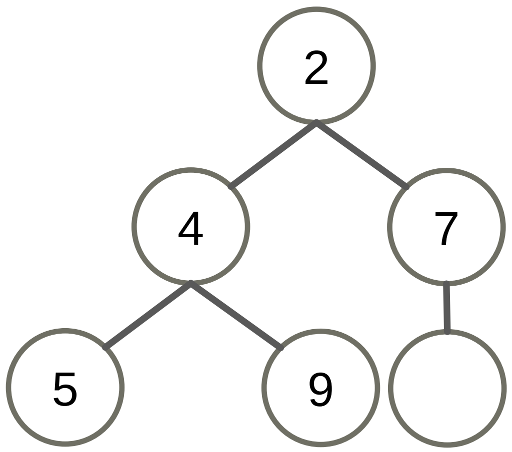
Dijkstra‘s search | working principle

- Dijkstra‘s search expands nodes according to a HEAP and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

Dijkstra‘s search | working principle

- Dijkstra‘s search expands nodes according to a HEAP and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

```
Min_Bin_Heap_Push(Node up)
    insert up at end of heap
    while up < parent(up):
        swap(up, parent(up))
```

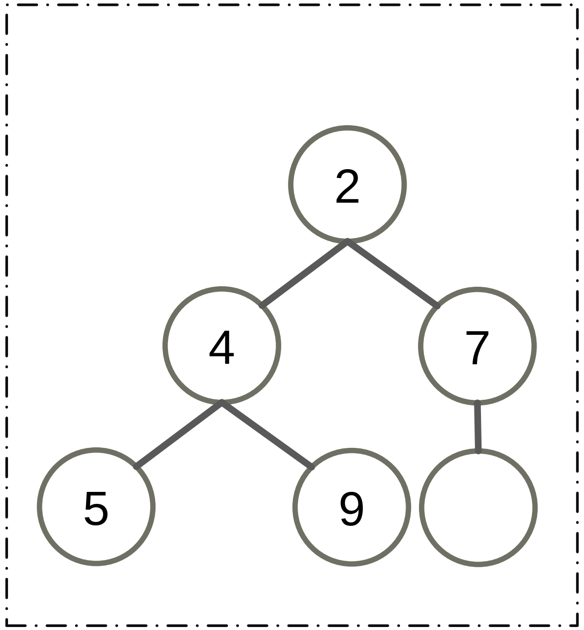


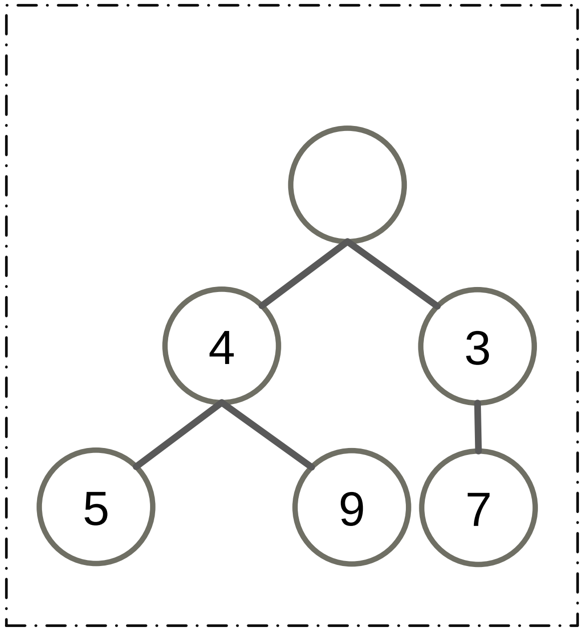
Dijkstra‘s search | working principle

- Dijkstra‘s search expands nodes according to a HEAP and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

```
Min_Bin_Heap_Push(Node up)
    insert up at end of heap
    while up < parent(up):
        swap(up, parent(up))

Min_Bin_Heap_Pop()
    return top element of heap
    move bottom element to top as down
    while down > min(child(down)):
        swap(down, min(child(down)))
```

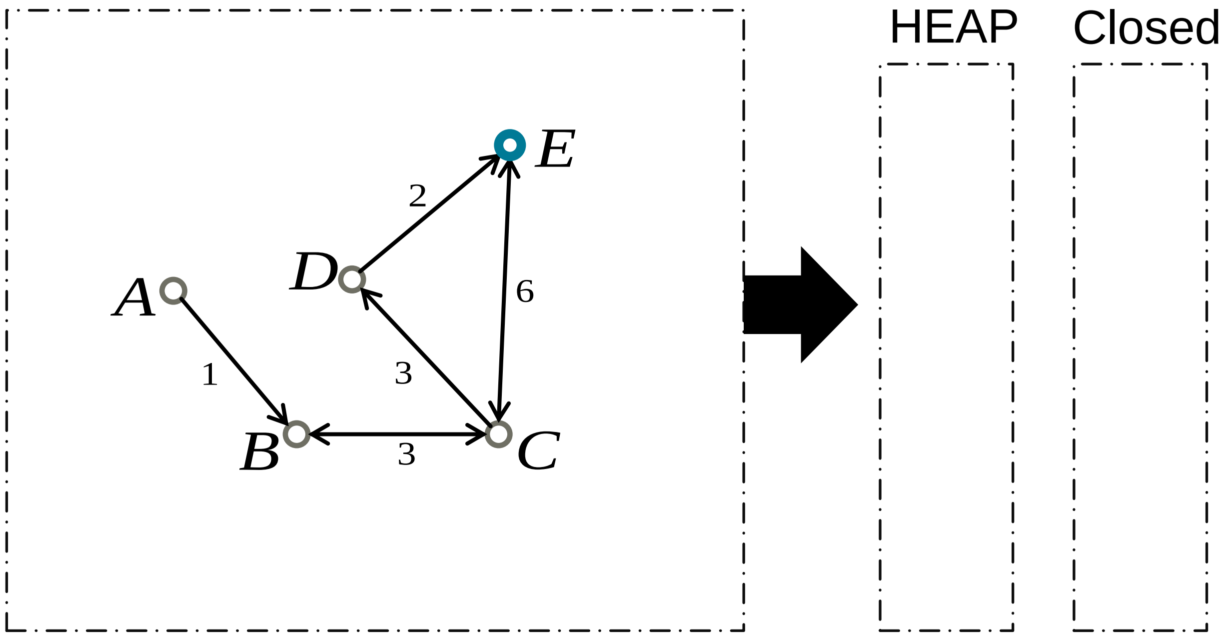




Dijkstra‘s search | working principle

- Dijkstra‘s search expands nodes according to a HEAP and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

```
Dijkstra(Graph G, Node Start, Node Goal)
    Queue.init(BIN_MIN_HEAP)
    Queue.push(Start)
    while Queue is not empty:
        Node curr = Queue.pop()
        if curr is Goal return
        Closed.push(curr)
        Nodes next = expand(curr)
        for all next not in Closed:
            Queue.push(next)
```



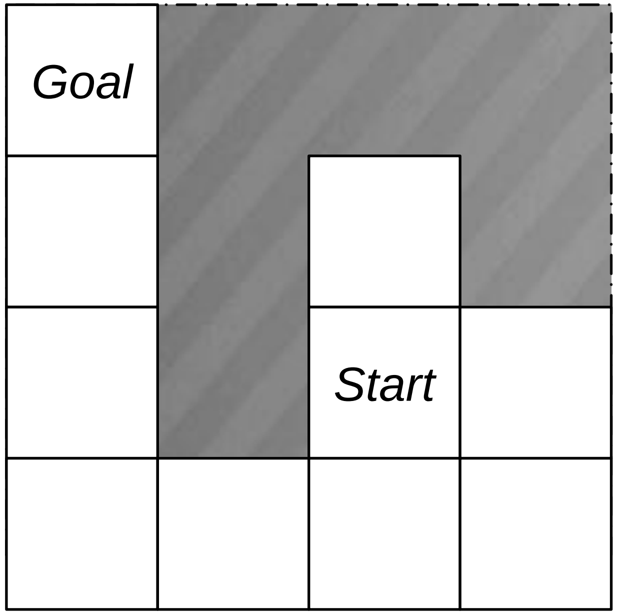
Dijkstra‘s search | properties & requirements

- The sequence to the first goal state encountered is optimal
- Edge costs must be strictly positive; otherwise, employ Bellman-Ford
- Dijkstra‘s search has a time complexity of $O(|V|\log|V|+|E|)$

The A* algorithm | working principle

- A* expands nodes according to a HEAP and a Closed list
- It makes use of a heuristic function to guide search
- It backtracks the solution from the goal state backwards in a greedy way

The A* algorithm | example




```
A_Star(Graph G, Heur H, Node Start, Node Goal)
    Queue.init(BIN_MIN_HEAP, H)
    Queue.push(Start)
    while Queue is not empty:
        Node curr = Queue.pop()
        if curr is Goal return
        Closed.push(curr)
        Nodes next = expand(curr)
        for all next not in Closed:
            Queue.push(next)
```

HEAP

Closed List

The A* algorithm | properties & requirements

- The trajectory to the first goal state encountered is optimal
- Edge costs must be strictly positive
- For optimality to hold heuristic must be consistent

Randomized graph search | overview

- Encompasses optimization algorithms operating according to a randomized node expansion step
- The associated graph is thus usually constructed online during search
- Randomization is appropriate for high-dimensional search spaces

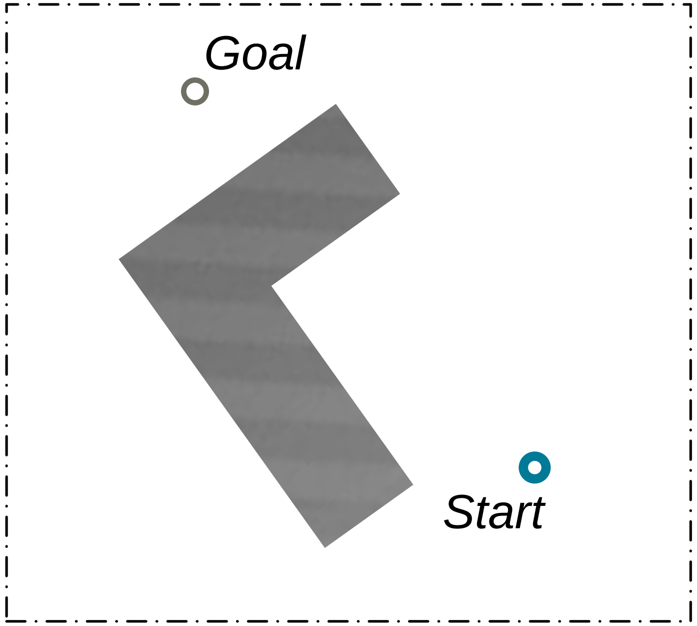
The RRT algorithm | working principle

- RRT grows a randomized tree during search
- It terminates once a state close to the goal state is expanded

The RRT algorithm | example

- RRT grows a randomized tree during search
- It terminates once a state close to the goal state is expanded

```
RRT(Node Start, Node Goal, System Sys,
        Environment Env)
    Graph.init(Start)
    while Graph.size() is less than threshold
        Node rand = rand()
        Node near = Graph.nearest(rand)
        try
            Node new  = Sys.propagate(near, rand)
            Graph.addNode(new)
            Graph.addEdge(near,new)
```



The RRT algorithm | properties

- Solutions are almost surely sub-optimal
- RRT is probabilistically complete

Graph search| further reading

- Any-angle search
  - D. Ferguson and A. T. Stentz. “Field D*: An Interpolation-based Path Planner and Replanner”. In *Proceedings of the International Symposium on Robotics Research (ISRR),* 2005.

- The D* algorithm
  - S. Koenig and M. Likhachev. “Improved Fast Replanning for Robot Navigation in Unknown Terrain”. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA),* 2002.

- The RRT* algorithm
  - S. Karaman and E. Frazzoli. “Sampling-based Algorithms for Optimal Motion Planning”. *International Journal of Robotics Research,* 30(7): 846–894, 2011.