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

\[
\text{BF}(\text{Graph G, Node Start, Node Goal})
\]

1. Queue.init(FIFO)
2. Queue.push(Start)
3. while Queue is not empty:
   4. Node curr = Queue.pop()
   5. if curr is Goal return
   6. Closed.push(curr)
   7. Nodes next = expand(curr)
   8. for all next not in Closed:
      9. 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 $O(|V| + |E|)$. 

---

The text is written in a natural style, and the mathematical notation is correctly transcribed. The properties of breadth-first search are clearly stated and explained.
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

0. **Min_Bin_Heap_Push**(Node up)
1. `insert` up at end of heap
3. `while` up < parent(up):
4. `swap`(up, parent(up))

Graph:
```
  2
 /\  
4 7
 /\  
5 9
```
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

0  Min_Bin_Heap_Push(Node up)
1  insert up at end of heap
3  while up < parent(up):
4   swap(up, parent(up))

0  Min_Bin_Heap.Pop()
1  return top element of heap
2  move bottom element to top as down
3  while down > min(child(down)):
4   swap(down, min(child(down)))

Graph Search | 8
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

```python
0 Dijkstra(Graph G, Node Start, Node Goal)
1  Queue.init(BIN_MIN_HEAP)
2  Queue.push(Start)
3  while Queue is not empty:
4      Node curr = Queue.pop()
5      if curr is Goal return
6      Closed.push(curr)
7      Nodes next = expand(curr)
8      for all next not in Closed:
9          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

0 \textbf{A\_Star}(\text{Graph G, Heur H, Node Start, Node Goal})
1 \hspace{1em} \text{Queue.init(BIN\_MIN\_HEAP, H)}
2 \hspace{1em} \text{Queue.push(Start)}
3 \hspace{1em} \textbf{while} \ \text{Queue is not empty:}
4 \hspace{2em} \text{Node curr = Queue.pop()}
5 \hspace{2em} \textbf{if} \ \text{curr is Goal return}
6 \hspace{2em} \text{Closed.push(curr)}
7 \hspace{2em} \text{Nodes next = expand(curr)}
8 \hspace{2em} \textbf{for all} \ \text{next not in} \ \text{Closed:}
9 \hspace{3em} \text{Queue.push(next)}
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

```
0 RRT(Node Start, Node Goal, System Sys, Environment Env)
1     Graph.init(Start)
2     while Graph.size() is less than threshold
3     Node rand = rand()
4     Node near = Graph.nearest(rand)
5     try
6     Node new = Sys.propagate(near, rand)
7     Graph.addNode(new)
8     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

- The D* algorithm

- The RRT* algorithm